从单片机初学者迈向单片机工程师

目录：

一、 LED 主题讨论周第一章----写在前面

学习单片机也已经有几年了，藉此机会和大家聊一下我学习过程中的一些经历和想法吧。也感谢一线工人提供了这个机会。希望大家有什么好的想法和建议都直接跟帖说出来。毕竟只有交流才能够碰撞出火花来^_^。

“卖弄”也好，“吹嘘”也罢，我只是想认真的写写我这一路走来历经的总总，把其中值得注意，以及经验的地方写出来，权当是我对自己的一个总结吧。而作为看官的你，如果看到了我的错误，还请一定指正，这样对我以及其它读者都有帮助，而至于你如果从中能够收获到些许，那便是我最大的欣慰了。姑妄言之，姑妄听之。如果有啥好的想法和建议一定要说出来。几年前，和众多初学者一样，我接触到了单片机，立刻被其神奇的功能所吸引，从此不能自拔。很多个日夜就这样陪伴着它度过了。期间也遇到过非常多的问题，也一度被这些问题所困惑……等到回过头来，看到自己曾经走过的路，唏嘘不已。经常混迹于论坛里，也看到了很多初学者发的求助帖子，看到他们走在自己曾走过的弯路上，忽然想到了自己的那段日子，心里竟然莫名的冲动，凡此总总，我总是尽自己所能去回帖。很多时候，都想写一点什么东西出来，希望对广大的初学者有一点点帮助。但总是不知从何处写起。今天借一线工人的台，唱一唱我的戏

一路学习过来的过程中，帮助最大之一无疑来自于网络了。很多时候，通过网络，我们都可以获取到所需要的学习资料。但是，随着我们学习的深入，我们会慢慢发现，网络提供的东西是有限度的，好像大部分的资料都差不多，或者说是适合大部分的初学者所需，而当我们想更进一步提高时，却发现能够获取到的资料越来越少，相信各位也会有同感，铺天盖地的单片机资料中大部分不是流水灯就是 LED，液晶，而且也只是仅仅作功能性的演示。于是有些人选择了放弃，或者是转移到其他兴趣上面去了，而只有少部分人选择了继续摸索下去，结合市面上的书籍，然后在网络上锲而不舍的搜集资料，再从牛人的只言片语中去体会，不断动手实践，慢慢的，也摸索出来了自己的一条路子。当然这个过程必然是艰辛的，而他学会了之后也不会在网络上轻易分享自己的学习成果。如此恶性循环下去，也就不难理解为什么初级的学习资料满天飞，而深入一点的学习资料却很少的原因了。相较于其他领域，单片机技术的封锁更加容易。尽管已经问世了很多年了，有价值的资料还是相当的欠缺，大部分的资料都是止于入门阶段或者是简单的演示实验。但是在实际工程应用中却是另外一回事。有能力的高手无暇或者是不愿公开自己的学习经验。

很多时候，我也很困惑，看到国外爱好者毫不保留的在网络上发布自己的作品，我忽然感觉到一丝丝的悲哀。也许，我们真的该转变一下思路了，帮助别人，其实也是在帮助自己。啰啰嗦嗦的说了这么多，相信大家能够明白说的是什么意思。在接下来的一段日子里，我将会结合电子工程师之家举办的主题周活动写一点自己的想法。尽可能从实用的角度去讲述。希望能够帮助更多的初学者更上一层楼。而关于这个主题周的最大主题我想了这样的一个名字“从单片机初学者迈向单片机工程师”。名字挺大挺响亮，给我的压力也挺大的，但我会努力，争取使这样的一系列文章能够带给大家一点帮助，而不是看后大跌眼镜。这样的一系列文章主要的对象是初学者，以及想从初学者更进一步提高的读者。而至于老手，以及那些牛 XX 的人，希望能够给我们这些初学者更多的一些指点哈~@_@~.

二、 LED 主题讨论周第二章----学会释放 CPU

从这一章开始，我们开始迈入单片机的世界。在我们开始这一章具体的学习之前，有必要给大家先说明一下。在以后的系列文章中，我们将以 51 内核的单片机为载体，C 语言为编程语言，开发环境为 KEIL uv3。至于为什么选用 C 语言开发，好处不言而喻，开发速度快，效率高，代码可复用率高，结构清晰，尤其是在大型的程序中，而且随着编译器的不断升级，其编译后的代码大小与汇编语言的差距越来越小。而关于 C 语言和汇编之争，就像那个啥，每隔一段时间总会有人挑起这个话题，如果你感兴趣，可以到网上搜索相关的帖子自行阅读。不是说汇编不重要，在很多对时序要求非常高的场合，需要利用汇编语言和 C 语言混合编程才能够满足系统的需求。在我们学习掌握 C 语言的同时，也还需要利用闲余的时间去学习了解汇编语言。

1.从点亮 LED(发光二极管)开始

在市面上众多的单片机学习资料中，最基础的实验无疑于点亮 LED 了，即控制单片机的 I/O 的电平的变化。如同如下实例代码一般

```
void main(void)
{
    LedInit() ;
    While(1)
    {
        LED = ON ;
        DelayMs(500) ;
        LED = OFF ;
        DelayMs(500) ;
}
}
```

程序很简单，从它的结构可以看出，LED 先点亮 500mS，然后熄灭 500mS，如此循环下去，形成的效果就是 LED 以 1HZ 的频率进行闪烁。下面让我们分析上面的程序有没有什么问题。

看来看出，好像很正常的啊，能有什么问题呢？这个时候我们应该换一个思路去想了。试想，整个程序除了控制 LED = ON ； LED = OFF； 这两条语句外，其余的时间，全消耗在了 DelayMs(500)这两个函数上。而在实际应用系统中是没有哪个系统只闪烁一只 LED 就其它什么事情都不做了的。因此，在这里我们要想办法，把 CPU 解放出来，让它不要白白浪费 500mS 的延时等待时间。宁可让它一遍又一遍的扫描看有哪些任务需要执行，也不要让它停留在某个地方空转消耗 CPU 时间。

从上面我们可以总结出

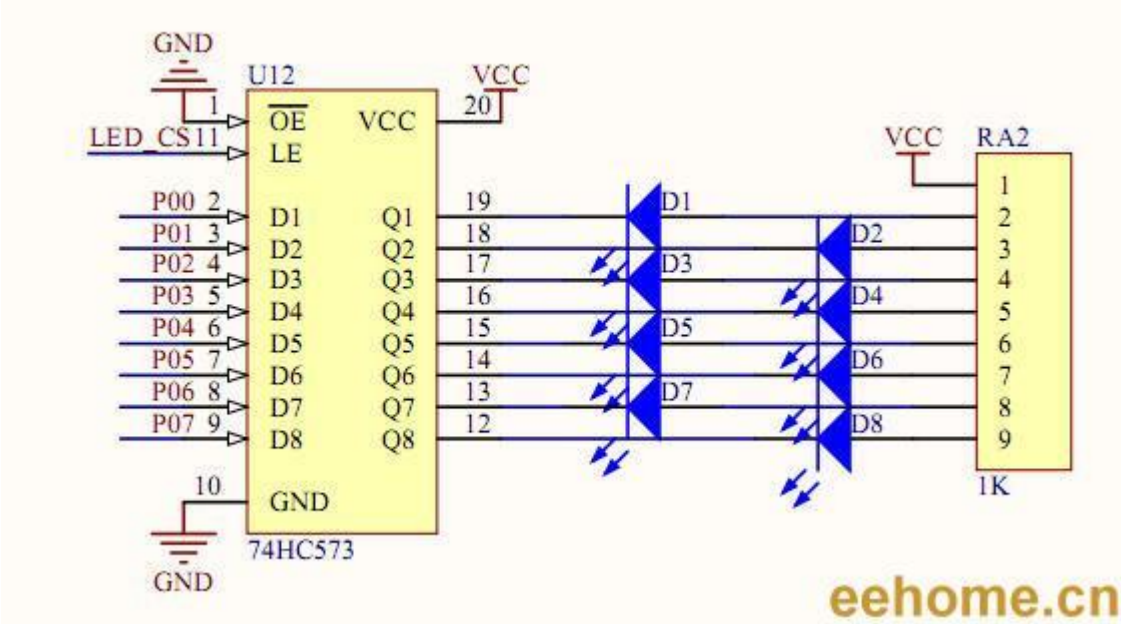
(1) 无论什么时候我们都要以实际应用的角度去考虑程序的编写。

(2) 无论什么时候都不要让 CPU 白白浪费等待，尤其是延时(超过 1MS)这样的地方。

下面让我们从另外一个角度来考虑如何点亮一颗 LED。

先看看我们的硬件结构是什么样子的。

我手上的单片机板子是电子工程师之家的开发的学习板。就以它的实际硬件连接图来分析吧。如下图所示



一般的 LED 的正常发光电流为 10~20mA 而低电流 LED 的工作电流在 2mA 以下（亮度与普通发光管相同）。在上图中我们可知，当 Q1~Q8 引脚上面的电平为低电平时，LED 发光。通过 LED 的电流约为（VCC - V_d）/ RA2 。其中 V_d 为 LED 导通后的压降，约为 1.7V 左右。这个导通压降根据 LED 颜色的不同，以及工作电流的大小的不同，会有一定的差别。下面一些参数是网上有人测出来的，供大家参考。

红色的压降为 1.82-1.88V，电流 5-8mA，
绿色的压降为 1.75-1.82V，电流 3-5mA，
橙色的压降为 1.7-1.8V，电流 3-5mA
兰色的压降为 3.1-3.3V，电流 8-10mA，
白色的压降为 3-3.2V，电流 10-15mA，
(供电电压 5V，LED 直径为 5mm)

74HC573 真值表如下：

FUNCTION TABLE

Inputs			Output
Output Enable	Latch Enable	D	Q
L	H	H	H
L	H	L	L
L	L	X	no change
H	X	X	Z

X = don't care
Z = high impedance

通过这个真值表我们可以看出。当 OutputEnable 引脚接低电平的时候，并且 LatchEnable 引脚为高电平的时候，Q 端电平与 D 端电平相同。结合我们的 LED 硬件连接图可以知道 LED_CS 端为高电平时候，P0 口电平的变化即 Q 端的电平的变化，进而引起 LED 的亮灭变化。由于单片机的驱动能力有限，在此，74HC573 的主要作用就是起一个输出驱动的作用。需要注意的是，通过 74HC573 的最大电流是有限制的，否则可能会烧坏 74HC573 这个芯片。

I_{OUT}	DC Output Current, per Pin	±35	mA
I_{CC}	DC Supply Current, V_{CC} and GND Pins	±75	mA

上面这个图是从 74HC573 的 DATASHEET 中截取出来的，从上可以看出，每个引脚允许通过的最大电流为 35mA 整个芯片允许通过的最大电流为 75mA。在我们设计相应的驱动电路时候，这些参数是相当重要的，而且是最容易被初学者所忽略的地方。同时在设计的时候，要留出一定量的余量出来，不能说单个引脚允许通过的电流为 35mA，你就设计为 35mA，这个时候你应该把设计的上限值定在 20mA 左右才能保证能够稳定的工作。

（设计相应驱动电路时候，应该仔细阅读芯片的数据手册，了解每个引脚的驱动能力，以及整个芯片的驱动能力）

了解了相应的硬件后，我们再来编写驱动程序。

首先定义 LED 的接口

```
#define LED   P0
```

然后为亮灭常数定义一个宏，由硬件连接图可以，当 P0 输出为低电平时候 LED 亮，P0 输出为高电平时，LED 熄灭。

```
#define LED_ON()     LED = 0x00    //所有 LED 亮
#define LED_OFF()    LED = 0xff    //所有 LED 熄灭
```

下面到了重点了，究竟该如何释放 CPU，避免其做延时空等待这样的事情呢。很简单，我们为系统产生一个 1mS 的时标。假定 LED 需要亮 500mS，熄灭 500mS，那么我们可以对这个 1mS 的时标进行计数，当这个计数值达到 500 时候，清零该计数值，同时把 LED 的状态改变。

```
unsigned int g_u16LedTimeCount = 0 ;     //LED 计数器
unsigned char g_u8LedState = 0 ;         //LED 状态标志, 0 表示亮，1 表示熄灭

void LedProcess(void)
{
    if(0 == g_u8LedState)  //如果 LED 的状态为亮，则点亮 LED
    {
        LED_ON() ;
    }
    else                   //否则熄灭 LED
    {
        LED_OFF() ;
    }
}


void LedStateChange(void)
{
```

```
    if(g_bSystemTime1Ms)                //系统 1MS 时标到
  {
        g_bSystemTime1Ms = 0 ;
        g_u16LedTimeCount++ ;       //LED 计数器加一
    if(g_u16LedTimeCount >= 500)   //计数达到 500,即 500MS 到了,改变 LED 的状态。
    {
            g_u16LedTimeCount = 0 ;
            g_u8LedState   = ! g_u8LedState ;
    }
  }
}
```

上面有一个变量没有提到，就是 g_bSystemTime1Ms 。这个变量可以定义为位变量或者是其它变量，在我们的定时器中断函数中对其置位，其它函数使用该变量后，应该对其复位(清 0) 。

我们的主函数就可以写成如下形式(示意代码)

```
void main(void)
{
    while(1)
    {
        LedProcess() ;
        LedStateChange() ;
    }
}
```

因为 LED 的亮或者灭依赖于 LED 状态变量(g_u8LedState)的改变，而状态变量的改变，又依赖于 LED 计数器的计数值(g_u16LedTimeCount ，只有计数值达到一定后，状态变量才改变)所以，两个函数都没有堵塞 CPU 的地方。让我们来从头到尾分析一遍整个程序的流程。

程序首先执行 LedProcess() ;函数

因为 g_u8LedState 的初始值为 0 (见定义，对于全局变量，在定义的时候最好给其一个确定的值)所以 LED 被点亮，然后退出 LedStateChange()函数，执行下一个函数 LedStateChange()

在函数 LedStateChange()内部首先判断 1mS 的系统时标是否到了，如果没有到就直接退出函数，如果到了，就把时标清 0 以便下一个时标消息的到来，同时对 LED 计数器加一，然后再判断 LED 计数器是否到达我们预先想要的值 500，如果没有，则退出函数，如果有，对计数器清 0，以便下次重新计数，同时把 LED 状态变量取反，然后退出函数。

由上面整个流程可以知道，CPU 所做的事情，就是对一些计数器加一，然后根据条件改变状态，再根据这个状态来决定是否点亮 LED。这些函数执行所花的时间都是相当短的，如果主程序中还有其它函数，则 CPU 会顺次往下执行下去。对于其它的函数(如果有的话)也要采取同样的措施，保证其不堵塞 CPU，如果全部基于这种方法设计，那么对于不是非常庞大的系统，我们的系统依旧可以保证多个任务(多个函数)同时执行。系统的实时性得到了一定的保证，从宏观上看来，就是多个任务并发执行。

好了，这一章就到此为止，让我们总结一下，究竟有哪些需要注意的吧。

(1) 无论什么时候我们都要以实际应用的角度去考虑程序的编写。
(2) 无论什么时候都不要让 CPU 白白浪费等待，尤其是延时(超过 1mS)这样的地方。
(3) 设计相应驱动电路时候，应该仔细阅读芯片的数据手册，了解每个引脚的驱动能力，以及整个芯片的驱动能力
(4) 最重要的是，如何去释放 CPU(参考本章的例子)，这是写出合格程序的基础。

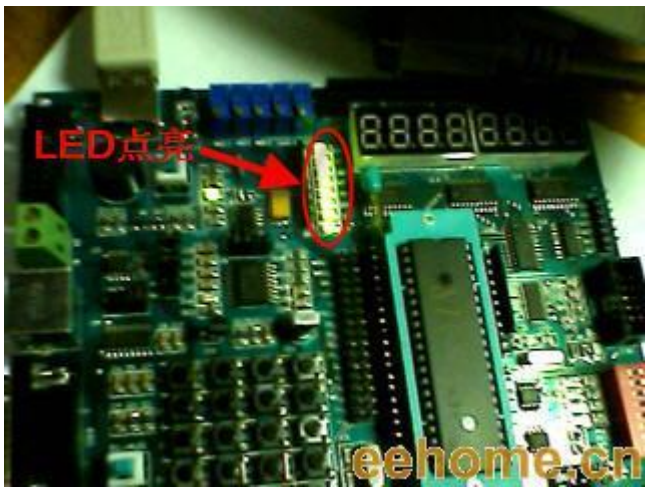
附完整程序代码(基于电子工程师之家的单片机开发板)

```
#include<reg52.h>

sbit LED_SEG   = P1^4;  //数码管段选
sbit LED_DIG   = P1^5;  //数码管位选
sbit LED_CS11 = P1^6;  //led 控制位
sbit ir=P1^7;
#define LED P0              //定义 LED 接口
bit   g_bSystemTime1Ms = 0 ;                // 1MS 系统时标
unsigned int   g_u16LedTimeCount = 0 ; //LED 计数器
unsigned char g_u8LedState = 0 ;       //LED 状态标志, 0 表示亮，1 表示熄灭

#define LED_ON()       LED = 0x00    //所有 LED 亮
#define LED_OFF()     LED = 0xff     //所有 LED 熄灭

void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;       //定时器 0 工作方式 1
    TH0   =   0xfc ;        //定时器初始值
    TL0   =   0x66 ;
    TR0   = 1 ;
    ET0   = 1 ;
}
void LedProcess(void)
{
    if(0 == g_u8LedState)  //如果 LED 的状态为亮，则点亮 LED
    {
        LED_ON() ;
    }
    else                    //否则熄灭 LED
    {
        LED_OFF() ;
    }
```

```
}

void LedStateChange(void)
{
    if(g_bSystemTime1Ms)              //系统 1mS 时标到
  {
            g_bSystemTime1Ms = 0 ;
            g_u16LedTimeCount++ ;       //LED 计数器加一
      if(g_u16LedTimeCount >= 500)   //计数达到 500,即 500mS 到了,改变 LED 的状态。
      {
      g_u16LedTimeCount = 0 ;
      g_u8LedState   = ! g_u8LedState     ;
          }
    }
}

void main(void)
{
    Timer0Init() ;
    EA = 1 ;
    LED_CS11 = 1 ; //74HC595 输出允许
    LED_SEG = 0 ;  //数码管段选和位选禁止(因为它们和 LED 共用 P0 口)
    LED_DIG = 0 ;
    while(1)
    {
          LedProcess() ;
          LedStateChange() ;
    }
}

void Time0Isr(void) interrupt 1
{
    TH0  =    0xfc ;              //定时器重新赋初值
    TL0  =  0x66 ;
    g_bSystemTime1Ms = 1 ;    //1MS 时标标志位置位
}
```

实际效果图如下

点亮



熄灭

三、 LED 主题讨论周第三章----模块化编程初识

好的开始是成功的一半!

通过上一章的学习，我想你已经掌握了如何在程序中释放 CPU 了。希望能够继续坚持下去。一个良好的开始是成功的一半。我们今天所做的一切都是为了在单片机编程上做的更好。

在谈论今天的主题之前，先说下我以前的一些经历。在刚开始接触到 C 语言程序的时候，由于学习内容所限，写的程序都不是很大，一般也就几百行而矣。所以所有的程序都完成在一个源文件里面。记得那时候大一参加学校里的一个电子设计大赛，调试了一个多星期，所有程序加起来大概将近 1000 行，长长的一个文件，从上浏览下来都要好半天。出了错误简单的语法错误还好定位，其它一些错误，往往找半天才找的到。那个时候开始知道了模块化编程这个东西，也尝试着开始把程序分模块编写。最开始是把相同功能的一些函数(譬如 1602 液晶的驱动)全部写在一个头文件(.h)文件里面，然后需要调用的地方包含进去，但是很快发现这种方法有其局限性，很容易犯重复包含的错误。而且调用起来也很不方便。很快暑假的电子设计大赛来临了，学校对我们的单片机软件编程进行了一些培训。由于学校历年来参加国赛和省赛，因此积累了一定数量的驱动模块，那些日子，老师每天都会布置一定量的任务，让我们用这些模块组合起来，完成一定功能。而正是那些日子模块化编程的培训，使我对于模块化编程有了更进一步的认识。并且程序规范也开始慢慢注意起来。此后的日子，无论程序的大小，均采用模块化编程的方式去编写。很长一段时间以来，一直有单片机爱好者在 QQ 上和我一起交流。有时候，他们会发过来一些有问题的程序源文件，让我帮忙修改一下。同样是长长的一个文件，而且命名极不规范，从头看下来，着实是痛苦，说实话，还真不如我重新给他们写一个更快一些，此话到不假，因为手头积累了一定量的模块，在完成一个新的系统时候，只需要根据上层功能需求，在底层模块的支持下，可以很快方便的完成。而不需要从头到尾再一砖一瓦的重新编写。藉此，也可以看出模块化编程的一个好处，就是可重复利用率高。下面让我们揭开模块化神秘面纱，一窥其真面目。

C 语言源文件 *.c

提到 C 语言源文件，大家都不会陌生。因为我们平常写的程序代码几乎都在这个 XX.C 文件里面。编译器也是以此文件来进行编译并生成相应的目标文件。作为模块化编程的组成基础，我们所要实现的所有功能的源代码均在这个文件里。理想的模块化应该可以看成是一个黑盒子。即我们只关心模块提供的功能，而不管模块内部的实现细节。好比我们买了一部手机，我们只需要会用手机提供的功能即可，不需要知晓它是如何把短信发出去的，如何响应我们按键的输入，这些过程对我们用户而言，就是是一个黑盒子。

在大规模程序开发中，一个程序由很多个模块组成，很可能，这些模块的编写任务被分配到不同的人。而你在编写这个模块的时候很可能就需要利用到别人写好的模块的接口，这个时候我们关心的是，它的模块实现了什么样的接口，我该如何去调用，至于模块内部是如何组织的，对于我而言，无需过多关注。而追求接口的单一性，把不需要的细节尽可能对外部屏蔽起来，正是我们所需要注意的地方。

C 语言头文件 *.h

谈及到模块化编程，必然会涉及到多文件编译，也就是工程编译。在这样的一个系统中，往往会有多个 C 文件，而且每个 C 文件的作用不尽相同。在我们的 C 文件中，由于需要对外提供接口，因此必须有一些函数或者是变量提供给外部其它文件进行调用。

假设我们有一个 LCD.C 文件，其提供最基本的 LCD 的驱动函数

```
LcdPutChar(char cNewValue) ;  //在当前位置输出一个字符
```

而在我们的另外一个文件中需要调用此函数，那么我们该如何做呢？

头文件的作用正是在此。可以称其为一份接口描述文件。其文件内部不应该包含任何实质性的函数代码。我们可以把这个头文件理解成为一份说明书，说明的内容就是我们的模块对外提供的接口函数或者是接口变量。同时该文件也包含了一些很重要的宏定义以及一些结构体的信息，离开了这些信息，很可能就无法正常使用接口函数或者是接口变量。但是总的原则是：不该让外界知道的信息就不应该出现在头文件里，而外界调用模块内接口函数或者是接口变量所必须的信息就一定要出现在头文件里，否则，外界就无法正确的调用我们提供的接口功能。因而为了让外部函数或者文件调用我们提供的接口功能，就必须包含我们提供的这个接口描述文件----即头文件。同时，我们自身模块也需要包含这份模块头文件(因为其包含了模块源文件中所需要的宏定义或者是结构体)，好比我们平常所用的文件都是一式三份一样，模块本身也需要包含这个头文件。

下面我们来定义这个头文件，一般来说，头文件的名字应该与源文件的名字保持一致，这样我们便可以清晰的知道哪个头文件是哪个源文件的描述。

于是便得到了 LCD.C 的头文件 LCD.h 其内容如下。

```
#ifndef    _LCD_H_
  #define  _LCD_H_
  extern   LcdPutChar(char cNewValue) ;
#endif
```

这与我们在源文件中定义函数时有点类似。不同的是，在其前面添加了 extern 修饰符表明其是一个外部函数，可以被外部其它模块进行调用。

```
#ifndef             _LCD_H_
        #define     _LCD_H_
#endif
```

这个几条条件编译和宏定义是为了防止重复包含。假如有两个不同源文件需要调用 LcdPutChar(char cNewValue) 这个函数，他们分别都通过 #include “Lcd.h” 把这个头文件包含了进去。在第一个源文件进行编译时候，由于没有定义过 _LCD_H_ 因此 #ifndef _LCD_H_ 条件成立，于是定义 _LCD_H_ 并将下面的声明包含进去。在第二个文件编译时候，由于第一个文件包含时候，已经将_LCD_H_定义过了。因此#ifndef _LCD_H_ 不成立，整个头文件内容就没有被包含。假设没有这样的条件编译语句，那么两个文件都包含了 extern LcdPutChar(char cNewValue) ; 就会引起重复包含的错误。

不得不说的 typedef

很多朋友似乎了习惯程序中利用如下语句来对数据类型进行定义

```
#define uint  unsigned int
#define uchar  unsigned char
```

然后在定义变量的时候 直接这样使用

```
uint  g_nTimeCounter = 0 ;
```

不可否认，这样确实很方便，而且对于移植起来也有一定的方便性。但是考虑下面这种情况你还会这么认为吗？

```
#define PINT unsigned int *  //定义 unsigned int 指针类型
PINT  g_npTimeCounter, g_npTimeState ;
```

那么你到底是定义了两个 **unsigned int** 型的指针变量，还是一个指针变量，一个整形变量呢？而你的初衷又是什么呢，想定义两个 **unsigned int** 型的指针变量吗？如果是这样，那么估计过不久就会到处抓狂找错误了。

庆幸的是 **C** 语言已经为我们考虑到了这一点。**typedef** 正是为此而生。为了给变量起一个别名我们可以用如下的语句

```
typedef unsigned int uint16;//给指向无符号整形变量起一个别名 uint16
typedef unsigned int * puint16;//给指向无符号整形变量指针起一个别名 puint16
```

在我们定义变量时候便可以这样定义了：

```
uint16    g_nTimeCounter  =  0 ;  //定义一个无符号的整形变量
puint16  g_npTimeCounter  ;     //定义一个无符号的整形变量的指针
```

在我们使用 **51** 单片机的 **C** 语言编程的时候，整形变量的范围是 **16** 位，而在基于 **32** 的微处理下的整形变量是 **32** 位。倘若我们在 **8** 位单片机下编写的一些代码想要移植到 **32** 位的处理器上，那么很可能我们就需要在源文件中到处修改变量的类型定义。这是一件庞大的工作，为了考虑程序的可移植性，在一开始，我们就应该养成良好的习惯，用变量的别名进行定义。

如在 **8** 位单片机的平台下，有如下一个变量定义

```
uint16    g_nTimeCounter  =  0 ;
```

如果移植 **32** 单片机的平台下，想要其的范围依旧为 **16** 位。

可以直接修改 **uint16** 的定义，即

```
typedef  unsigned  short  int    uint16 ;
```

这样就可以了，而不需要到源文件处处寻找并修改。

将常用的数据类型全部采用此种方法定义，形成一个头文件，便于我们以后编程直接调用。

文件名 **MacroAndConst.h**

其内容如下：

```
#ifndef   _MACRO_AND_CONST_H_
#define   _MACRO_AND_CONST_H_

typedef    unsigned int   uint16;
typedef    unsigned int   UINT;
typedef    unsigned int   uint;
typedef    unsigned int   UINT16;
typedef    unsigned int   WORD;
typedef    unsigned int   word;
typedef      int          int16;
typedef      int          INT16;
```

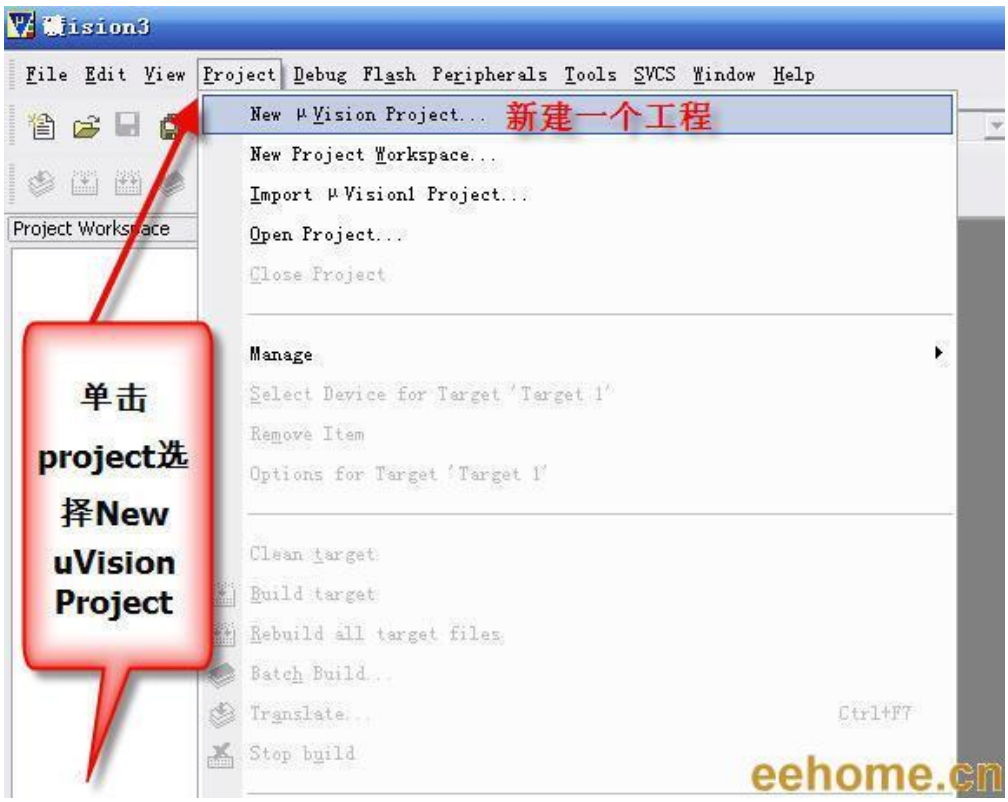
```
typedef    unsigned long  uint32;

typedef    unsigned long  UINT32;
typedef    unsigned long  DWORD;
typedef    unsigned long  dword;
typedef    long           int32;
typedef    long           INT32;
typedef    signed  char   int8;
typedef    signed  char   INT8;
typedef    unsigned char  byte;
typedef    unsigned char  BYTE;
typedef    unsigned char  uchar;
typedef    unsigned char  UINT8;
typedef    unsigned char  uint8;
typedef    unsigned char  BOOL;

#endif
```

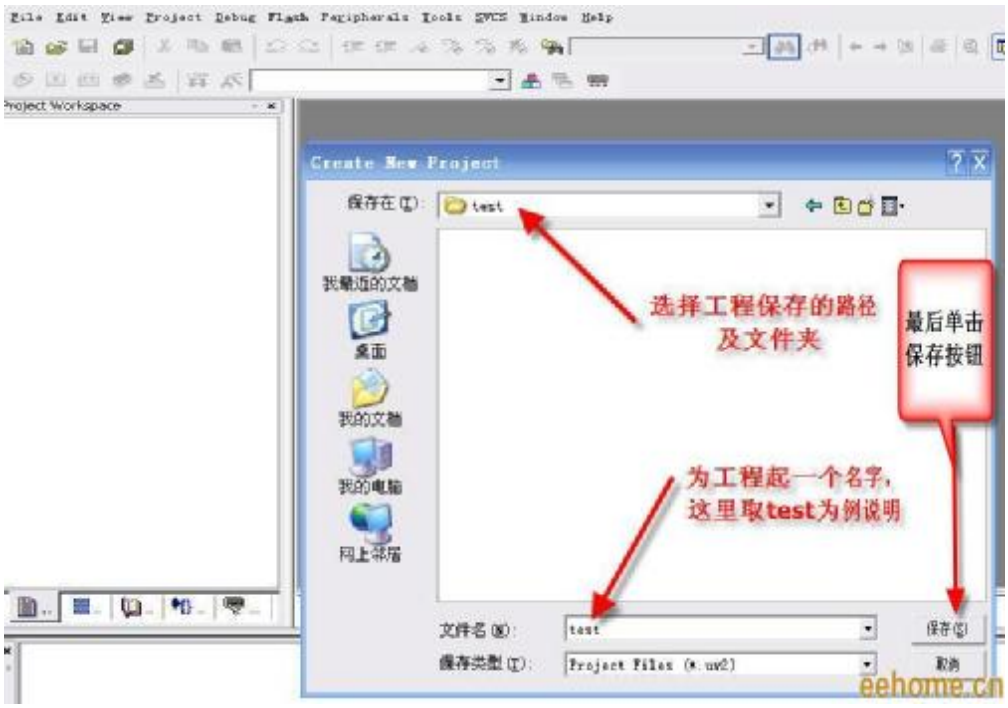
至此，似乎我们对于源文件和头文件的分工以及模块化编程有那么一点概念了。那么让我们趁热打铁，将上一章的我们编写的 **LED** 闪烁函数进行模块划分并重新组织进行编译。

在上一章中我们主要完成的功能是 **P0** 口所驱动的 **LED** 以 **1Hz** 的频率闪烁。其中用到了定时器，以及 **LED** 驱动模块。因而我们可以简单的将整个工程分成三个模块，定时器模块，**LED** 模块，以及主函数
对应的文件关系如下

```
main.c
Timer.h◊Timer.c  --
Led.h◊Led.c      --
```

在开始重新编写我们的程序之前，先给大家讲一下如何在 **KEIL** 中建立工程模板吧，这个模板是我一直沿用至今。希望能够给大家一点启发。
下面的内容就主要以图片为主了。同时辅以少量文字说明。
我们以芯片 **AT89S52** 为例。

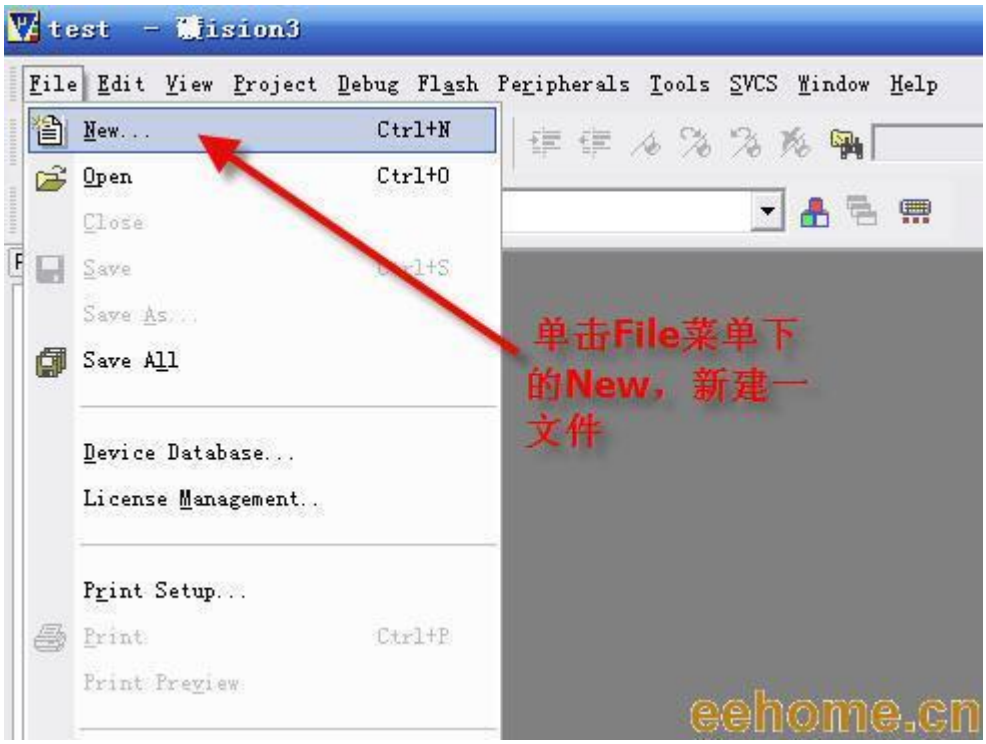

Vision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
New μVision Project...
新建一个工程
New Project Workspace...
Import μVision1 Project...
Open Project...
Close Project
Manage
Select Device for Target 'Target 1'
Remove Item
Options for Target 'Target 1'
Clean target
Build target
Rebuild all target files
Batch Build...
Translate... Ctrl+F7
Stop build
Project Workspace
单击project选择New uVision Project
eehome.cn

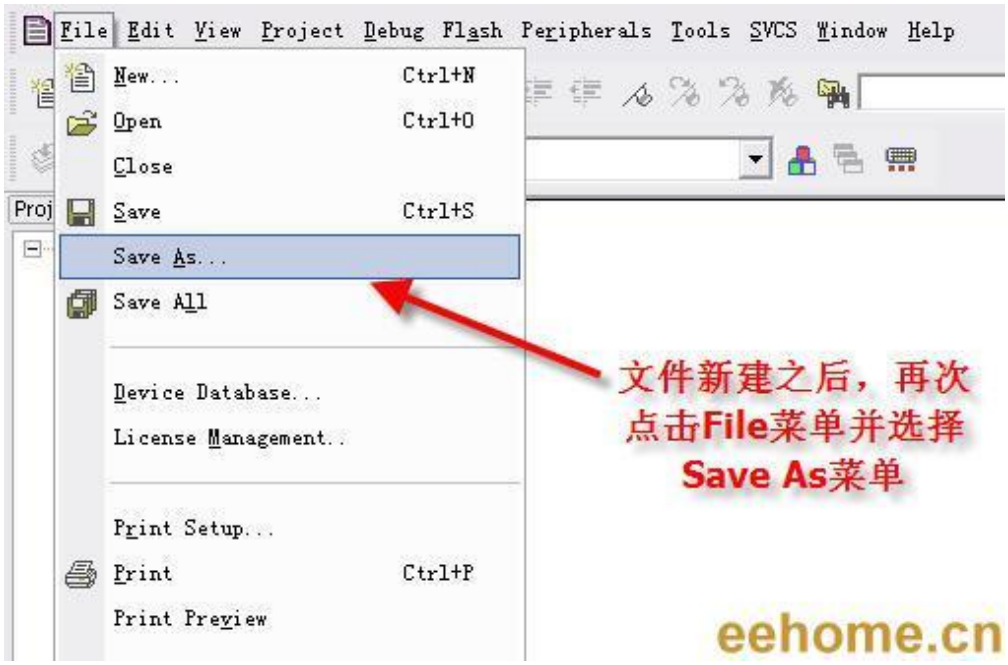

Create New Project
保存在(I):
test
选择工程保存的路径及文件夹
最后单击保存按钮
为工程起一个名字，这里取test为例说明
文件名(N):
test
保存(S)
保存类型(T):
Project Files (*.uv2)
取消
eehome.cn


Target 1
Select Device for Target 'Target 1'
CPU
Vendor: Acer Labs
Device:
Toolset:
Data base
Description
在弹出来的对话框中，通过滚动
条找到Atmel并展开其子项
Atmel
80C32E
AT48801
AT90S2X2
AT80C31X2
AT83/87C5103
AT83/87C5111
AT83/87C5112
AT83C5134
AT83C5135
AT83C5136
OK
Cancel
Help
eehome.cn

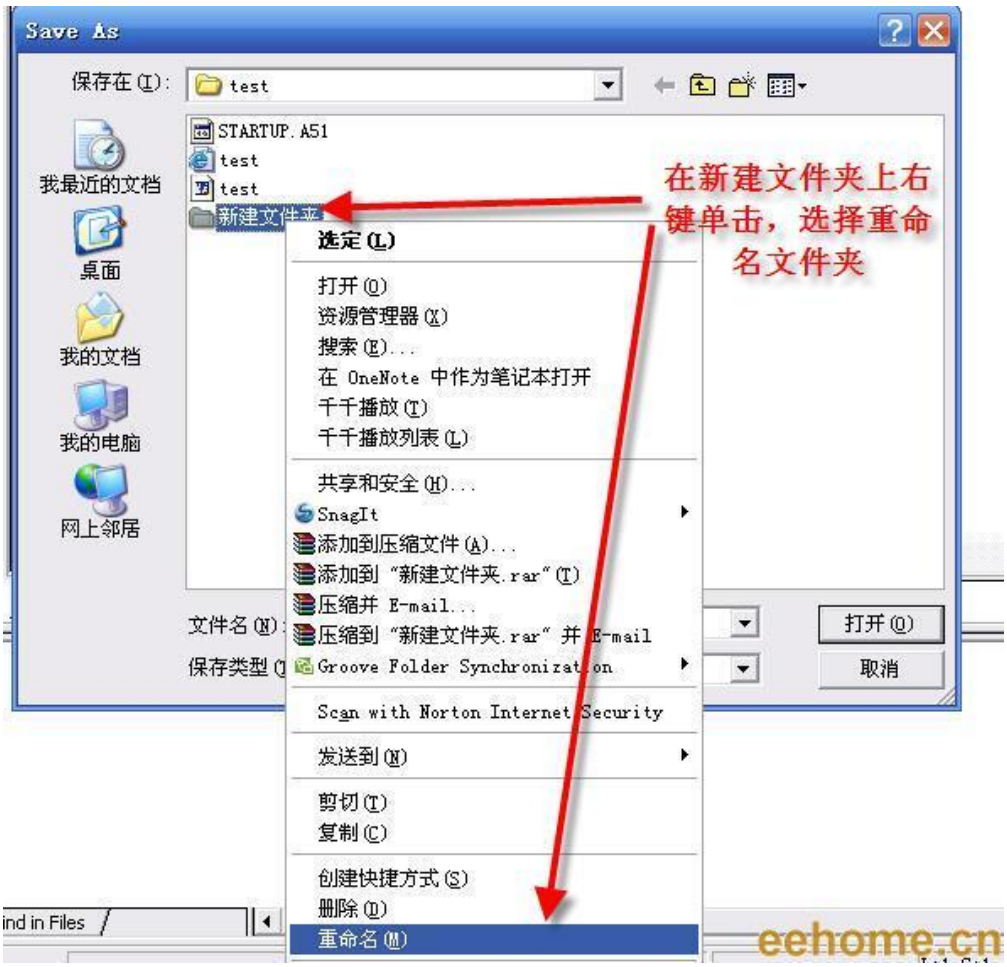

Select Device for Target 'Target 1'
CPU
在子项中选中AT89S52
Vendor:
Device:
Toolset: C51
Data base
Description
AT89LV55
AT89S2051
AT89S4051
AT89S51
AT89S52
AT89S53
AT89S8252
AT89S8253
AT8xC5122
T80C31
T80C31X2
T80C32
8051 based Full Static CMOS controller with Three-Level Program Memory Lock, 32 I/O lines, 3 Timers/Counters, 8 Interrupts Sources, Watchdog Timer, 2 DPTRs, 8K Flash Memory, 256 Bytes On-chip RAM
然后单击OK按钮
OK
Cancel
eehome.cn


Target 1
在弹出来的对话框中，选择是，将启动代码添加到工程中
Vision
Copy Standard 8051 Startup Code to Project Folder and Add File to Project ?
是(Y)
否(N)
eehome.cn


test - Vision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
New... Ctrl+N
Open Ctrl+O
Close
Save
Save As...
Save All
Device Database...
License Management...
Print Setup...
Print Ctrl+P
Print Preview
单击File菜单下的New，新建一文件
eehome.cn


File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
New... Ctrl+N
Open Ctrl+O
Close
Save Ctrl+S
Save As...
Save All
Device Database...
License Management...
Print Setup...
Print Ctrl+P
Print Preview
文件新建之后，再次
点击File菜单并选择
Save As菜单
eehome.cn


Target 1
Save As
保存在(I): test
STARTUP.A51
test
test
新建文件夹
我最近的文档
桌面
我的文档
我的电脑
网上邻居
在弹出的保存文件对话框
中，单击新建文件夹按钮，
新建一个空的文件夹
文件名(N):
保存类型(T): All Files (*.*)
打开(O)
取消
eehome.cn

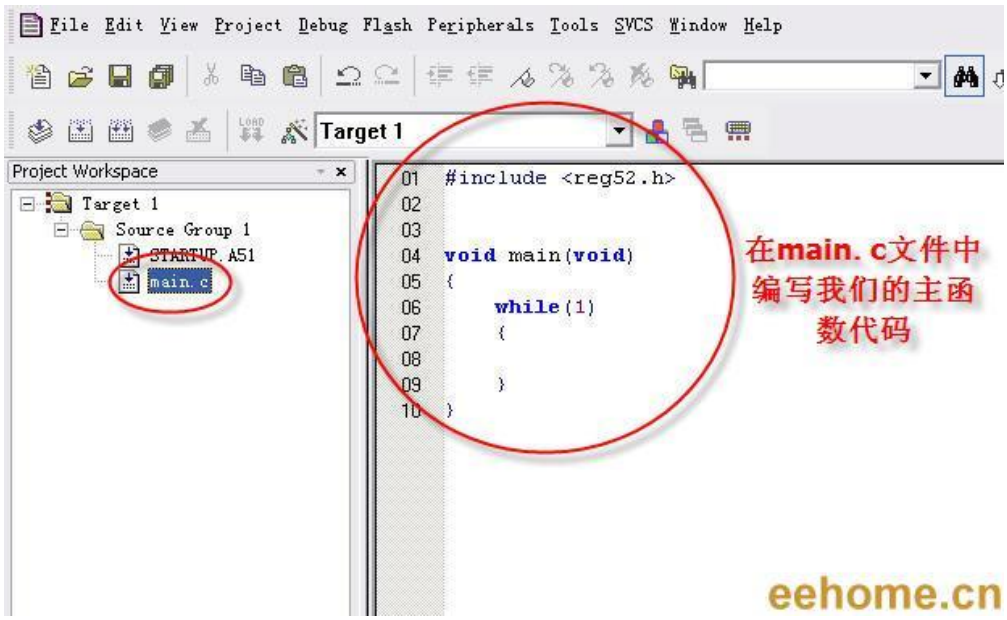

Save As
保存在(I):
test
STARTUP.A51
test
test
新建文件夹
我最近的文档
桌面
我的文档
我的电脑
网上邻居
选定(L)
打开(O)
资源管理器(X)
搜索(E)...
在 OneNote 中作为笔记本打开
千千播放(T)
千千播放列表(L)
共享和安全(H)...
SnagIt
添加到压缩文件(A)...
添加到 "新建文件夹.rar"(T)
压缩并 E-mail...
压缩到 "新建文件夹.rar" 并 E-mail
Groove Folder Synchronization
Scan with Norton Internet Security
发送到(N)
剪切(T)
复制(C)
创建快捷方式(S)
删除(D)
重命名(M)
文件名(N):
保存类型(T):
打开(O)
取消
在新建文件夹上右键单击，选择重命名文件夹
eehome.cn

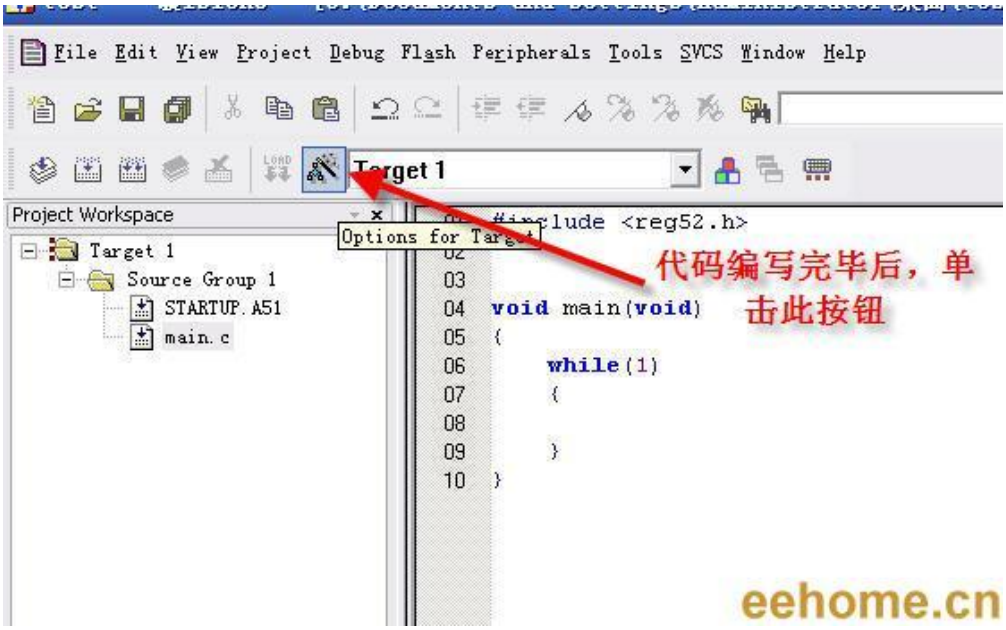

Save As
保存在(I):
test
STARTUP.A51
test
test
src
我最近的文档
桌面
我的文档
我的电脑
网上邻居
文件名(N):
保存类型(T):
All Files (*.*)
打开(O)
取消
将文件夹重命名为src
双击进入文件夹
eehome.cn

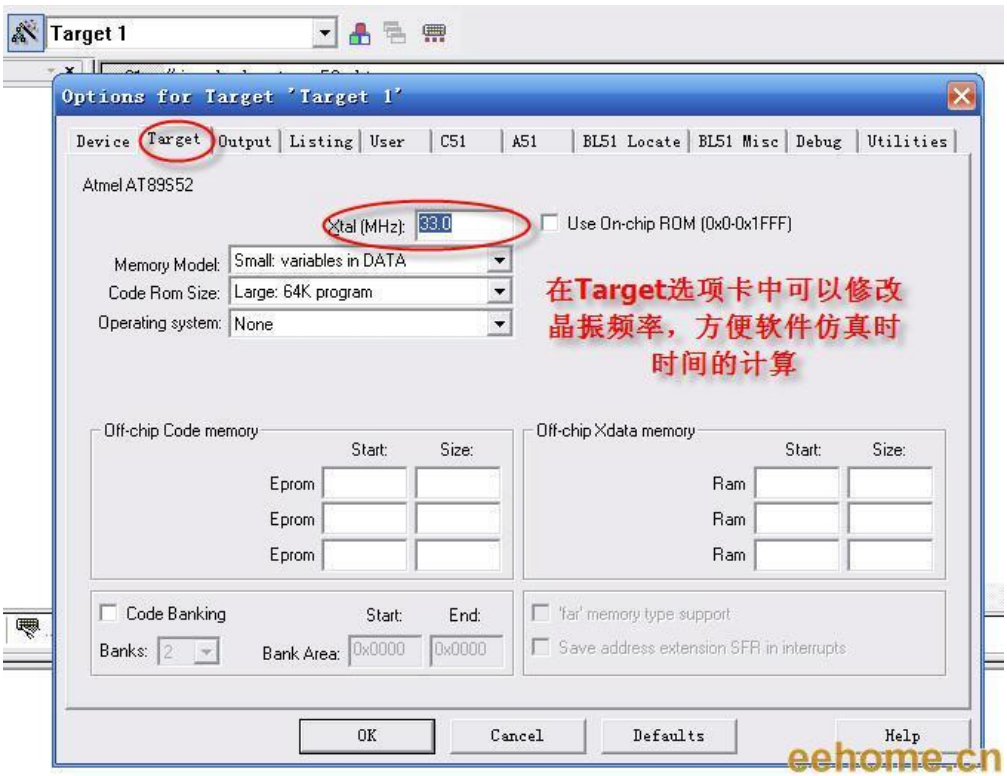

Save As
保存在(I):
src
我最近的文档
桌面
我的文档
我的电脑
网上邻居
给文件取一个名字为main. c, 然后单击保存按钮
文件名(N):
main.c
保存(S)
保存类型(T):
All Files (*.*)
取消
eehome.cn

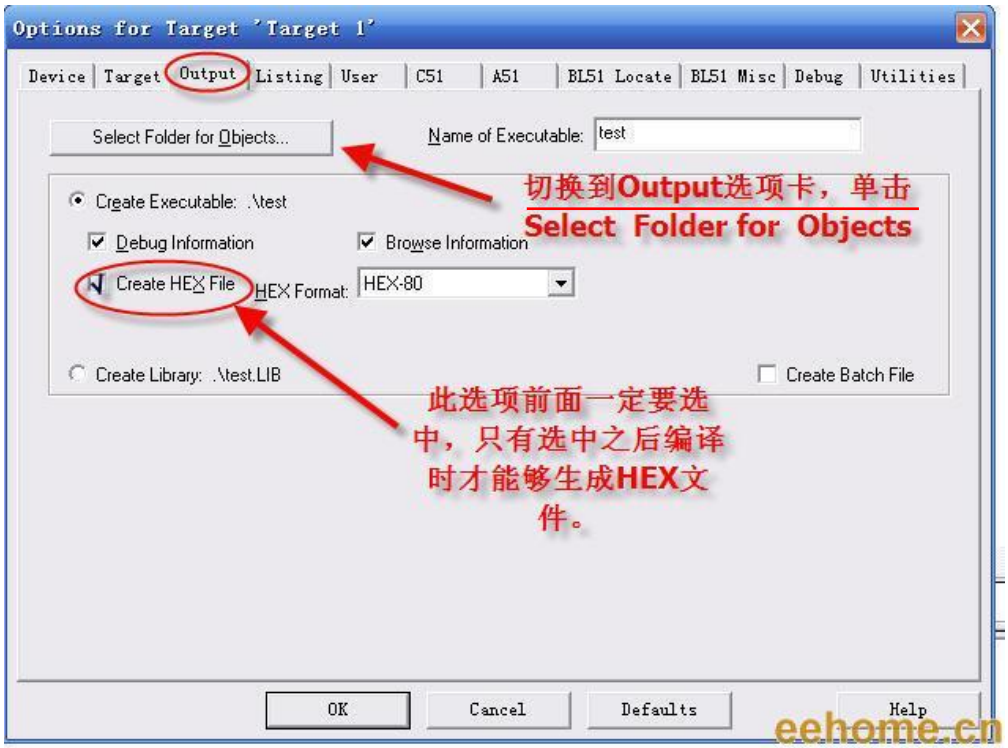

File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
STARTUP.A
在Source Group1上单击鼠标右键，在弹出的菜单中选择Add Files to Group 'Source Group 1'
Options for Group 'So
Open List File
Open Map File
Open File
Rebuild all target files
Build target
Translate File
Stop build
New Gro
Add Files to Group 'Source Group 1'
Manage Components
Remove Group 'Source Group 1' and its Files
Include Dependencies
eehome.cn

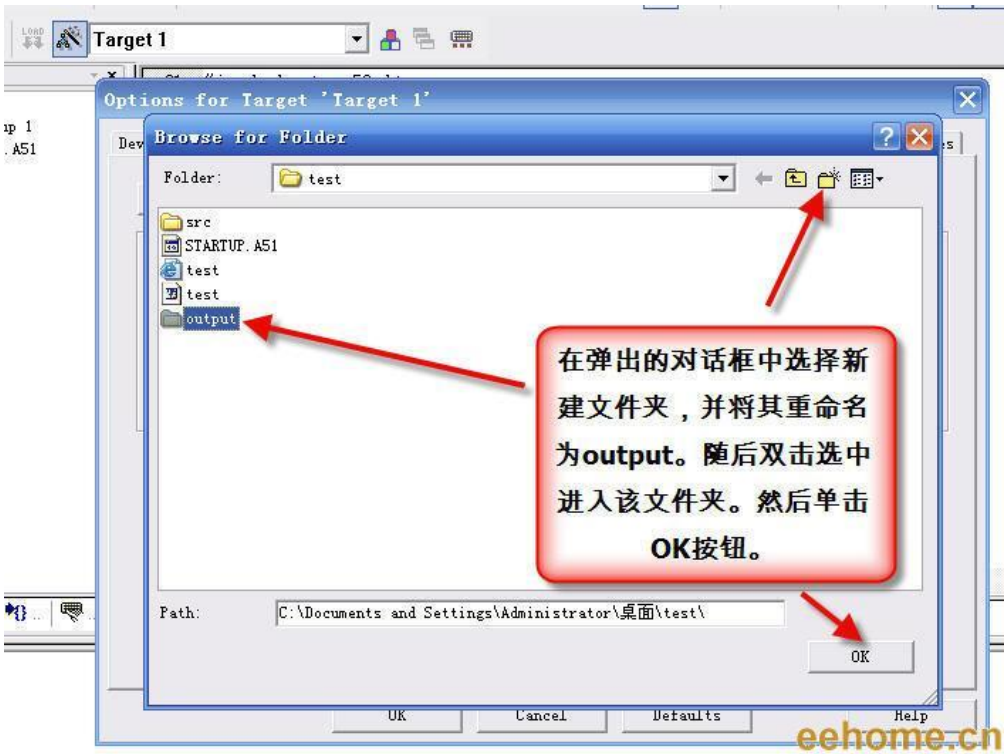

Add Files to Group 'Source Group 1'
查找范围(I):
src
main.c
选中我们之前新建的src文件夹，然后选中main.c文件，单击Add按钮后，再单击Close按钮
文件名(N):
main
文件类型(T):
C Source file (*.c)
Add
Close
eehome.cn

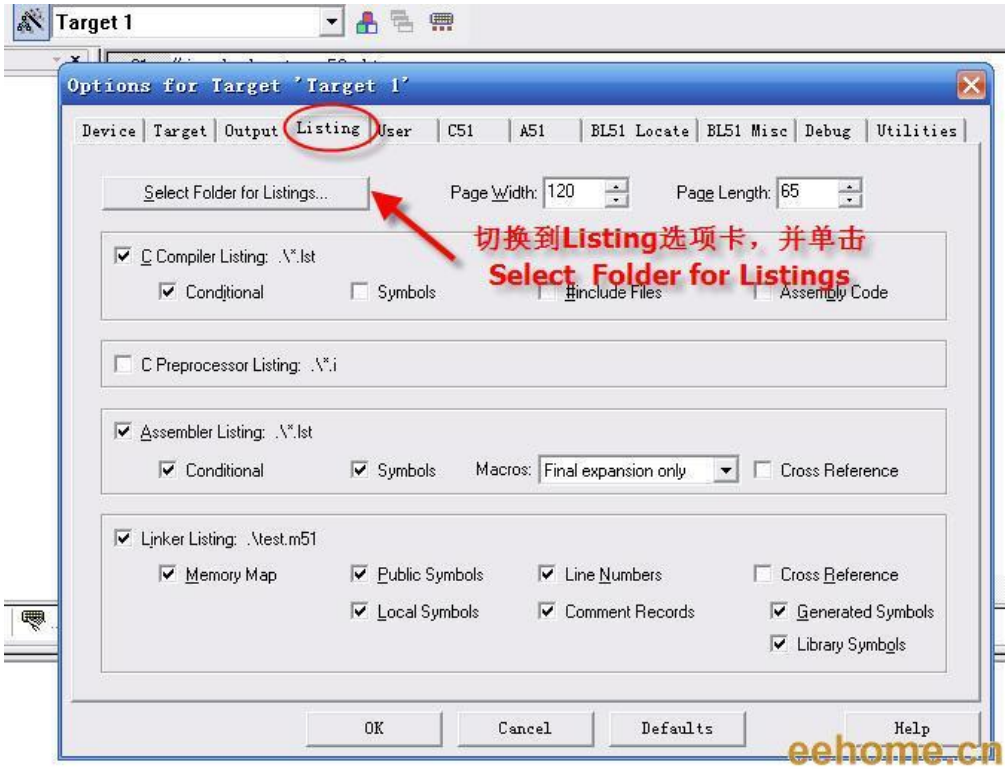

File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
STARTUP.A51
main.c
#include <reg52.h>
void main(void)
{
while(1)
{
}
}
在main.c文件中编写我们的主函数代码
eehome.cn


File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Options for Target
Target 1
Source Group 1
STARTUP.A51
main.c
#include <reg52.h>
void main(void)
{
while(1)
{
}
}
代码编写完毕后，单击此按钮
eehome.cn

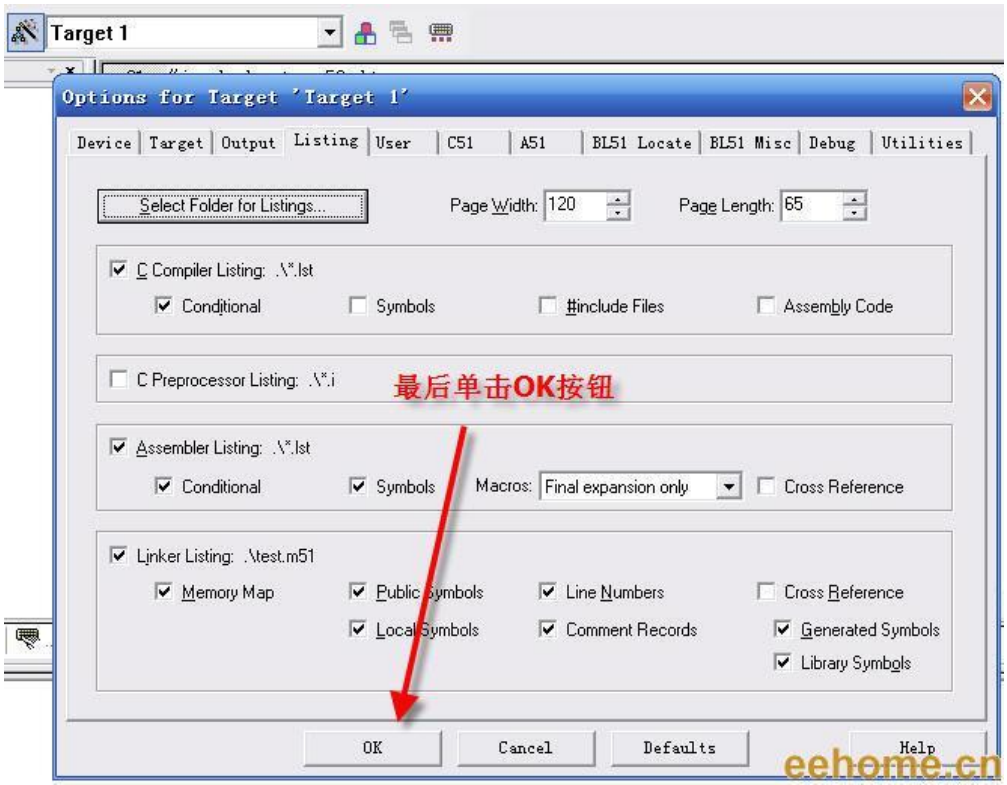

Target 1
Options for Target 'Target 1'
Device Target Output Listing User C51 A51 BL51 Locate BL51 Misc Debug Utilities
Atmel AT89S52
Xtal (MHz): 33.0
Use On-chip ROM (0x0-0x1FFF)
Memory Model: Small: variables in DATA
Code Rom Size: Large: 64K program
Operating system: None
在Target选项卡中可以修改晶振频率，方便软件仿真时时间的计算
Off-chip Code memory
Start: Size:
Eprom
Eprom
Eprom
Off-chip Xdata memory
Start: Size:
Ram
Ram
Ram
Code Banking
Start: End:
Banks: 2
Bank Area: 0x0000 0x0000
'far' memory type support
Save address extension SFR in interrupts
OK
Cancel
Defaults
Help
eehome.cn

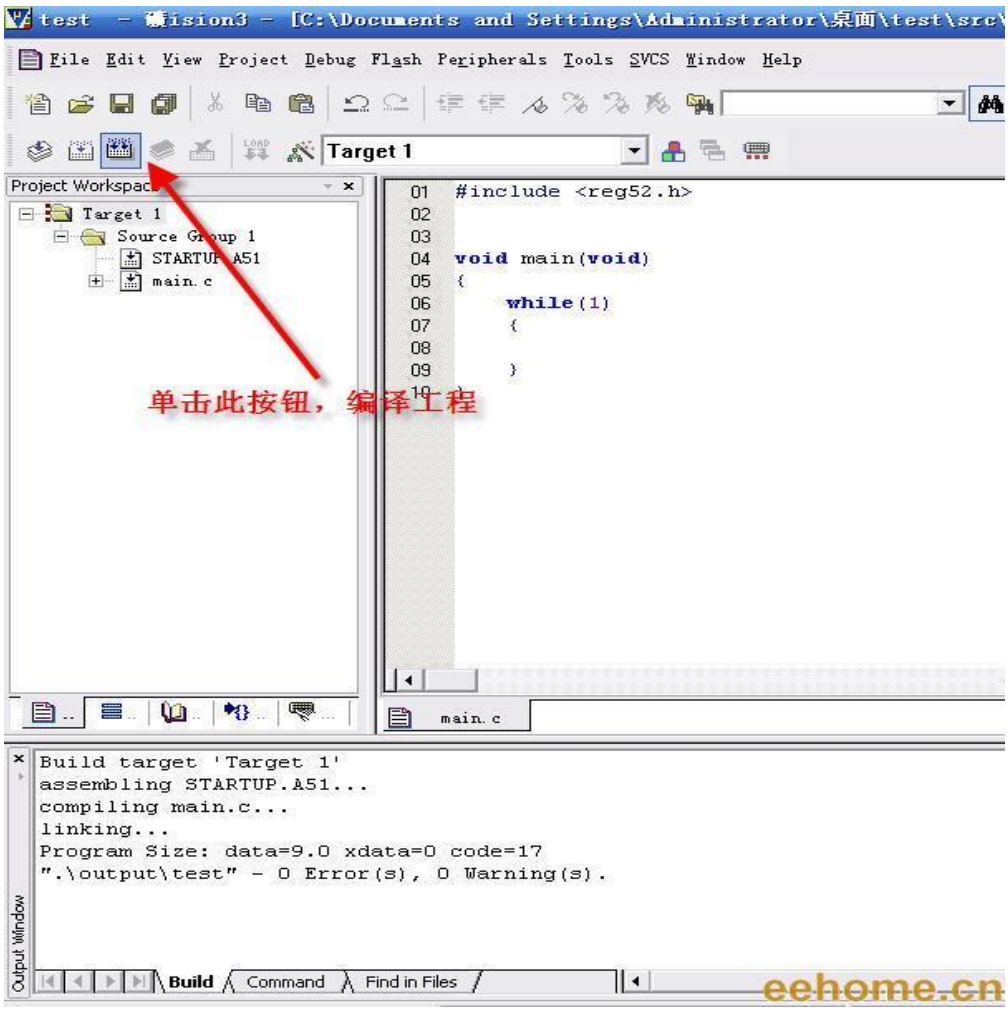

Options for Target 'Target 1'
Device
Target
Output
Listing
User
C51
A51
BL51 Locate
BL51 Misc
Debug
Utilities
Select Folder for Objects...
Name of Executable: test
切换到Output选项卡，单击
Select Folder for Objects
Create Executable: .\test
Debug Information
Browse Information
Create HEX File
HEX Format: HEX-80
Create Library: .\test.LIB
Create Batch File
此选项前面一定要选中，只有选中之后编译时才能够生成HEX文件。
OK
Cancel
Defaults
Help
eehome.cn


Target 1
Options for Target 'Target 1'
Browse for Folder
Folder: test
src
STARTUP.A51
test
test
output
在弹出的对话框中选择新建文件夹，并将其重命名为output。随后双击选中进入该文件夹。然后单击OK按钮。
Path: C:\Documents and Settings\Administrator\桌面\test\
OK
OK
Cancel
Defaults
Help
eehome.cn


Target 1
Options for Target 'Target 1'
Device
Target
Output
Listing
User
C51
A51
BL51 Locate
BL51 Misc
Debug
Utilities
Select Folder for Listings...
Page Width: 120
Page Length: 65
切换到Listing选项卡，并单击
Select Folder for Listings
C Compiler Listing: .*.lst
Conditional
Symbols
#include Files
Assembly Code
C Preprocessor Listing: .*.i
Assembler Listing: .*.lst
Conditional
Symbols
Macros: Final expansion only
Cross Reference
Linker Listing: .\test.m51
Memory Map
Public Symbols
Line Numbers
Cross Reference
Local Symbols
Comment Records
Generated Symbols
Library Symbols
OK
Cancel
Defaults
Help
eehome.cn


Options for Target 'Target 1'
Browse for Folder
Folder:
test
output
src
STARTUP.A51
test
test
双击选中output文件夹，
然后单击OK按钮
Path:
C:\Documents and Settings\Administrator\桌面\test\
OK
OK
Cancel
Defaults
Help
eehome.cn


Target 1
Options for Target 'Target 1'
Device
Target
Output
Listing
User
C51
A51
BL51 Locate
BL51 Misc
Debug
Utilities
Select Folder for Listings...
Page Width: 120
Page Length: 65
C Compiler Listing: .*.lst
Conditional
Symbols
#include Files
Assembly Code
C Preprocessor Listing: .*.i
最后单击OK按钮
Assembler Listing: .*.lst
Conditional
Symbols
Macros: Final expansion only
Cross Reference
Linker Listing: .\test.m51
Memory Map
Public Symbols
Line Numbers
Cross Reference
Local Symbols
Comment Records
Generated Symbols
Library Symbols
OK
Cancel
Defaults
Help
eehome.cn


test - 礦ision3 - [C:\Documents and Settings\Administrator\桌面\test\src
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
STARTUP.A51
main.c
01 #include <reg52.h>
02
03
04 void main(void)
05 {
06 while(1)
07 {
08
09 }
10 }
单击此按钮，编译工程
main.c
Build target 'Target 1'
assembling STARTUP.A51...
compiling main.c...
linking...
Program Size: data=9.0 xdata=0 code=17
".\output\test" - 0 Error(s), 0 Warning(s).
Output Window
Build
Command
Find in Files
eehome.cn

OK ，到此一个简单的工程模板就建立起来了，以后我们再新建源文件和头文件的时候，就可以直接保存到 src 文件目录下面了。

下面我们开始编写各个模块文件。

首先编写 Timer.c 这个文件主要内容就是定时器初始化，以及定时器中断服务函数。其内容如下。

```
#include <reg52.h>

bit g_bSystemTime1Ms = 0 ;                  // 1mS 系统时标


void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;       //定时器 0 工作方式 1
    TH0  =  0xfc ;       //定时器初始值
    TL0  =  0x66 ;
    TR0  = 1 ;
    ET0  = 1 ;
}

void Time0Isr(void) interrupt 1
{
    TH0  =  0xfc ;              //定时器重新赋初值
    TL0  =  0x66 ;
    g_bSystemTime1Ms = 1 ;     //1mS 时标标志位置位
}
```

由于在 Led.c 文件中需要调用我们的 g_bSystemTime1Ms 变量。同时主函数需要调用 Timer0Init()初始化函数，所以应该对这个变量和函数在头文件里作外部声明。以方便其它函数调用。

Timer.h 内容如下。

```
#ifndef _TIMER_H_
#define _TIMER_H_

extern void Timer0Init(void) ;
extern bit g_bSystemTime1Ms ;

#endif
```

完成了定时器模块后，我们开始编写 LED 驱动模块。

Led.c 内容如下：

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Led.h"
#include "Timer.h"

static uint16   g_u16LedTimeCount = 0 ; //LED 计数器
static uint8   g_u8LedState = 0 ;       //LED 状态标志, 0 表示亮，1 表示熄灭

#define LED P0                  //定义 LED 接口
#define LED_ON()        LED = 0x00     //所有 LED 亮
#define LED_OFF()      LED = 0xff      //所有 LED 熄灭

void LedProcess(void)
{
    if(0 == g_u8LedState)   //如果 LED 的状态为亮，则点亮 LED
    {
        LED_ON() ;
    }
    else                    //否则熄灭 LED
    {
        LED_OFF() ;
    }
}


void LedStateChange(void)
{
    if(g_bSystemTime1Ms)                //系统 1mS 时标到
    {
        g_bSystemTime1Ms = 0 ;
        g_u16LedTimeCount++ ;       //LED 计数器加一
```

```
        if(g_u16LedTimeCount >= 500) //计数达到 500,即 500mS 到了,改变 LED 的状态。
        {
            g_u16LedTimeCount = 0 ;
            g_u8LedState   = ! g_u8LedState     ;
        }
    }
}
```

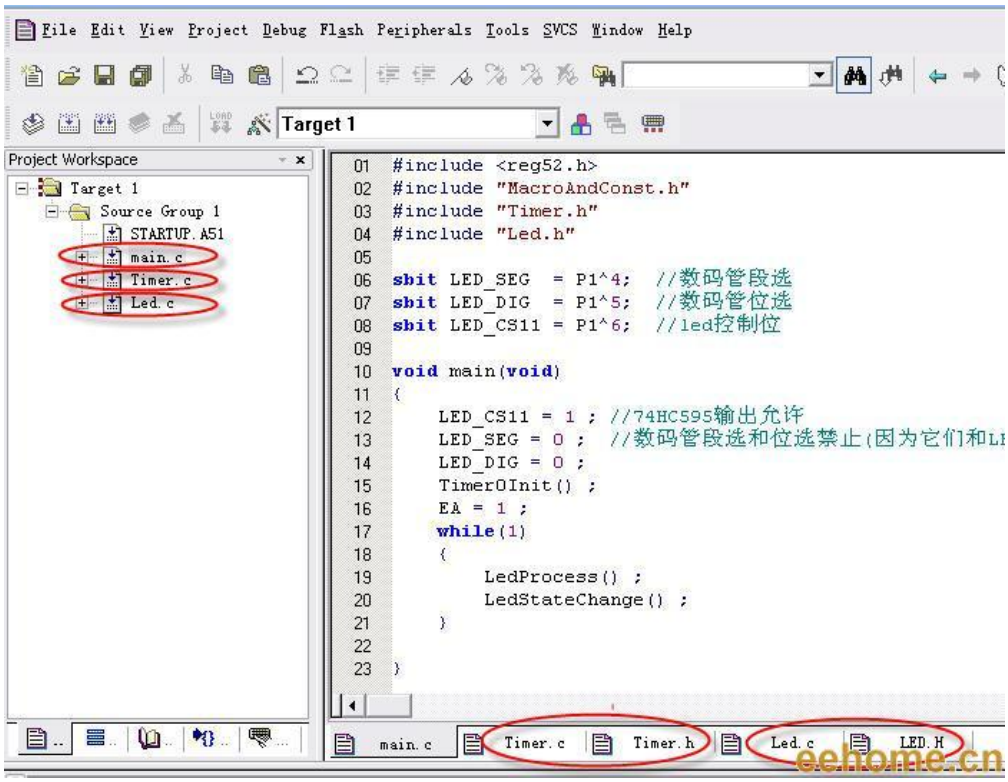
这个模块对外的借口只有两个函数，因此在相应的 Led.h 中需要作相应的声明。
Led.h 内容：

```
#ifndef _LED_H_
#define _LED_H_

extern void LedProcess(void) ;
extern void LedStateChange(void) ;

#endif
```

这两个模块完成后，我们将其 C 文件添加到工程中。然后开始编写主函数里的代码。
如下所示：

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Timer.h"
#include "Led.h"

sbit LED_SEG   = P1^4;   //数码管段选
sbit LED_DIG   = P1^5;   //数码管位选
sbit LED_CS11 = P1^6;   //led 控制位

void main(void)
{
        LED_CS11 = 1 ; //74HC595 输出允许
        LED_SEG = 0 ;   //数码管段选和位选禁止(因为它们和 LED 共用 P0 口)
        LED_DIG = 0 ;
        Timer0Init() ;
        EA = 1 ;
        while(1)
      {
       LedProcess() ;
       LedStateChange() ;
      }
}
```

整个工程截图如下：



至此，第三章到此结束。
一起来总结一下我们需要注意的地方吧

1. **C 语言源文件(*.c)的作用是什么**
2. **C 语言头文件(*.h)的作用是什么**
3. **typedef 的作用**
4. **工程模板如何组织**
5. **如何创建一个多模块(多文件)的工程**

四、 LED 主题讨论周第四章----渐明渐暗的灯

看着学习板上的 LED 按照我们的意愿开始闪烁起来，你心里是否高兴了，我相信你会的。但是很快你就会感觉到太单调，总是同一个频率在闪烁，总是同一个亮度在闪烁。如果要是能够由暗逐渐变亮，然后再由亮变暗该多漂亮啊。嗯，想法不错，可以该从什么地方入手呢。

在开始我们的工程之前，首先来了解一个概念：PWM。

PWM(Pulse Width Modulation)是脉冲宽度调制的英文单词的缩写。下面这段话是通信百科中对其的定义：脉冲宽度调制(PWM)是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。脉宽调制是开关型稳压电源中的术语。这是按稳压的控制方式分类的，除了 PWM 型，还有 PFM 型和 PWM、PFM 混合型。脉宽调制式开关型稳压电路是在控制电路输出频率不变的情况下，通过电压反馈调整其占空比，从而达到稳定输出电压的目的。

读起来有点晦涩难懂。其实简单的说来，PWM 技术就是通过调整一个周期固定的方波的占空比，来调节输出电压的平均当电压，电流或者功率等被控量。我们可以用一个水龙头来类比，把 1S 时间分成 50 等份，即每一个等份 20mS。在这 20mS 时间里如果我们把水龙头水阀一直打开，那么在这 20mS 里流过的水肯定是最多的，如果我们把水阀打开 15mS，剩下的 5mS 关闭水阀，那么流出的水相比刚才 20mS 全开肯定要小的多。同样的道理，我们可以通过控制 20mS 时间里水阀开启的时间的长短来控制流过的水的多少。那么在 1S 内平均流出的水流量也就可以被控制了。

当我们调整 PWM 的占空比时，就会引起电压或者电流的改变，LED 的明暗状态就会随之发生相应的变化，听起来好像可以通过这种方法来实现我们想要的渐明渐暗的效果。让我们来试一下吧。

大家都知道人眼有一个临界频率，当 LED 的闪烁频率达到一定的时候，人眼就分辨不出 LED 是否在闪烁了。就像我们平常看电视一样，看起来画面是连续的，实质不是这个样子，所有连续动作都是一帧帧静止的画面在 1S 的时间里快速播放出来，譬如每秒 24 帧的速度播放，由于人眼的视觉暂留效应，看起来画面就是连续的了。同样的道理，为了让我们的 LED 在变化的过程中，我们感觉不到其在闪烁，可以将其闪烁的频率定在50Hz以上。同时为了看起来明暗过渡的效果更加明显，我们在这里定义其变化范围为0~99（100等分）.即最亮的时候其灰度等级为 99，为 0 的时候最暗，也就是熄灭了。

于是乎我们定义 PWM 的占空比上限为 99, 下限定义为 0

```
#define   LED_PWM_LIMIT_MAX     99
#define   LED_PWM_LIMIT_MIN      0
```

假定我们 LED 的闪烁频率为 50HZ，而亮度变化的范围为 0~99 共 100 等分。则每一等分所占用的时间为 1/(50*100) = 200us 即我们在改变 LED 的亮灭状态时，应该是在 200us 整数倍时刻时。在这里我们用单片机的定时器产生 200us 的中断，同时每 20MS 调整一次 LED 的占空比。这样在 20mS * 100 = 2S 的时间内 LED 可以从暗逐渐变亮，在下一个 2S 内可以从亮逐渐变暗，然后不断循环。

由于大部分的内容都可以在中断中完成，因此，我们的大部分代码都在 Timer.c 这个文件中编写，主函数中除了初始化之外，就是一个空的死循环。

Timer.c 内容如下:

```
#include <reg52.h>
#include "MacroAndConst.h"

#define LED P0                  //定义 LED 接口
#define LED_ON()        LED = 0x00     //所有 LED 亮
#define LED_OFF()      LED = 0xff     //所有 LED 熄灭

#define   LED_PWM_LIMIT_MAX      99
#define   LED_PWM_LIMIT_MIN        0

static uint8 s_u8TimeCounter = 0 ; //中断计数
static uint8 s_u8LedDirection = 0 ; //LED 方向控制  0 ： 渐亮  1 ： 渐灭
static int8   s_s8LedPWMCounter   = 0 ; //LED 占空比
void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;      //定时器 0 工作方式 1
    TH0   =    0xff ;       //定时器初始值(200us 中断一次)
    TL0   =   0x47 ;
    TR0   = 1 ;
    ET0   = 1 ;
}

void Time0Isr(void) interrupt 1
{
    static int8 s_s8PWMCounter = 0   ;
    TH0   =   0xff ;              //定时器重新赋初值
    TL0   =   0x47 ;

    if(++s_u8TimeCounter >= 100) //每 20mS 调整一下 LED 的占空比
    {
        s_u8TimeCounter = 0 ;
        //如果是渐亮方向变化,则占空比递增
         if((s_s8LedPWMCounter <= LED_PWM_LIMIT_MAX) &&(0 == s_u8LedDirection))
        {
            s_s8LedPWMCounter++   ;
            if(s_s8LedPWMCounter > LED_PWM_LIMIT_MAX)
            {
                s_u8LedDirection = 1 ;
                s_s8LedPWMCounter =   LED_PWM_LIMIT_MAX ;
            }
        }
```

```
    //如果是渐暗方向变化,则占空比递渐
        if((s_s8LedPWMCounter >= LED_PWM_LIMIT_MIN) &&(1 == s_u8LedDirection))
        {
            s_s8LedPWMCounter-- ;
            if(s_s8LedPWMCounter < LED_PWM_LIMIT_MIN)
            {
                s_u8LedDirection = 0 ;
                s_s8LedPWMCounter =   LED_PWM_LIMIT_MIN ;
            }
        }
        s_s8PWMCounter = s_s8LedPWMCounter ;   //获取 LED 的占空比
    }

    if(s_s8PWMCounter > 0) //占空比大于 0,则点亮 LED,否则熄灭 LED
    {
        LED_ON() ;
        s_s8PWMCounter-- ;
    }
    else
    {
        LED_OFF();
    }

}
```

其实 PWM 技术在我们实际生活中应用的非常多。比较典型的应用就是控制电机的转速，控制充电电流的大小，等等。而随着技术的发展，也出现了其他类型的 PWM 技术，如相电压 PWM，线电压 PWM，SPWM 等等，如果有兴趣可以到网上去获取相应资料学习。

关于渐明渐暗的灯就简单的讲到这里。

五、 LED 主题讨论周第五章----多任务环境下的数码管编程设计

数码管在实际应用中非常广泛，尤其是在某些对成本有限制的场合。编写一个好用的 LED 程序并不是那么的简单。曾经有人这样说过，如果用数码管和按键，做一个简易的可以调整的时钟出来，那么你的单片机就算入门了 60%了。此话我深信不疑。我遇到过很多单片机的爱好者，他们问我说单片机我已经掌握了，该如何进一步的学习下去呢？我并不急于回答他们的问题，而是问他们：会编写数码管的驱动程序了吧？“嗯”。会编写按键程序了吧？“嗯”。好，我给你出一个小题目，你做一下。用按键和数码管以及单片机定时器实现一个简易的可以调整的时钟，要求如下：

8 位数码管显示，显示格式如下

时-分-秒

XX-XX-XX

要求：系统有四个按键，功能分别是 调整，加，减，确定。在按下调整键时候，显示时的两位数码管以 1 Hz 频率闪烁。如果再次按下调整键，则分开闪烁，依次循环，直到按下确定键，恢复正常的显示。在数码管闪烁的时候，按下加或者减键可以调整相应的显示内容。按键支持短按，和长按，即短按时，修改的内容每次增加一或者减小一，长按时候以一定速率连续增加或者减少。

结果很多人，很多爱好者一下子都理不清楚思路。其实问题的根源在于没有以工程化的角度去思考程序的编写。很多人在学习数码管编程的时候，都是照着书上或者网上的例子来进行试验。殊不知，这些例子代码仅仅只是具有一个演示性的作用，拿到实际中是很难用的。举一个简单的例子。

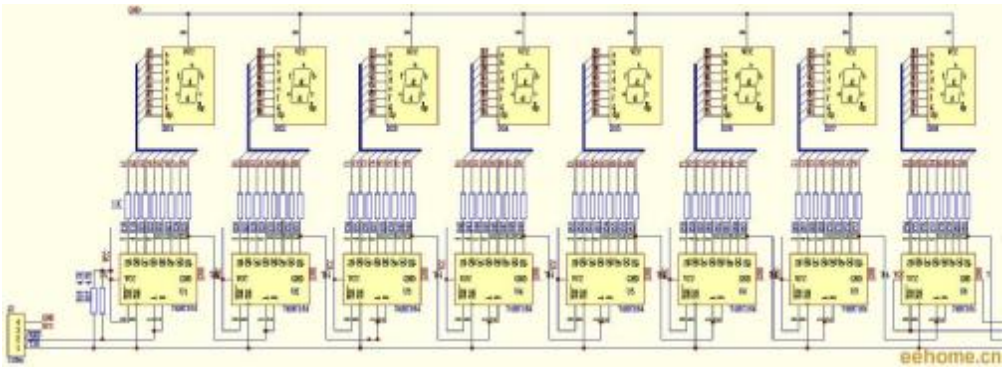
下面这段程序是在网上随便搜索到的：

```
while(1)
{
    for(num=0;num<9;num++)
    {
            P0=table[num];
            P2=code[num] ;
            delayms(2) ;
    }
}
```

看出什么问题来了没有，如果没有看出来请仔细想一下，如果还没有想出来，请回过头去，认真再看一遍“学会释放 CPU”这一章的内容。这个程序作为演示程序是没有什么问题的，但是实际应用的时候，数码管显示的内容经常变化，而且还有很多其它任务需要执行，因此这样的程序在实际中是根本就无法用的，更何况，它这里也调用了 delayms(2)这个函数来延时 2 ms 这更是令我们深恶痛绝

本章的内容正是探讨如何解决多任务环境下(不带 OS)的数码管程序设计的编写问题。理解了其中的思想，无论要求我们显示的形式怎么变化(如数码管闪烁，移位等),我们都可以很方便的解决问题。

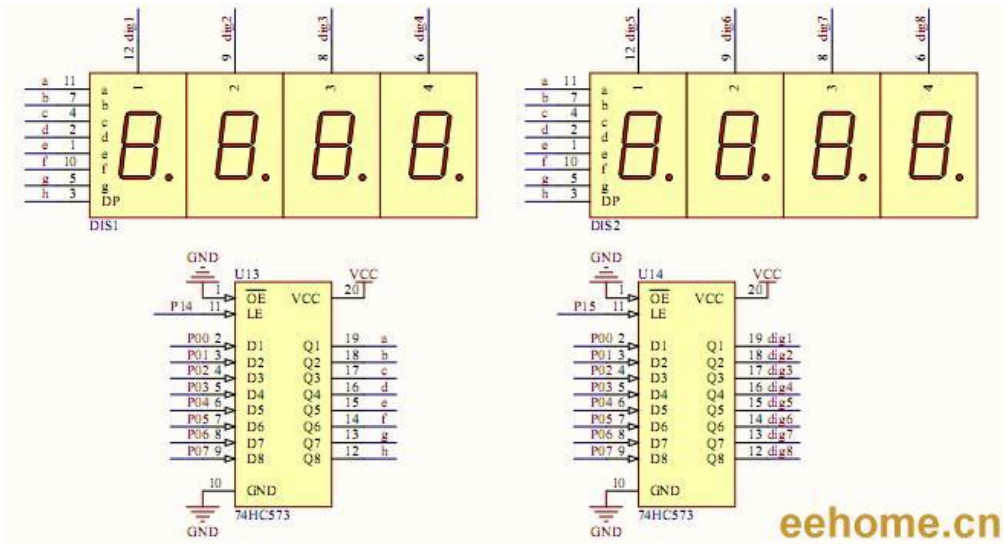
数码管的显示分为动态显示和静态显示两种。静态显示是每一位数码管都用一片独立的驱动芯片进行驱动。比较常见的有 74LS164，74HC595 等。利用这类芯片的好处就是可以级联，留给单片机的接口只需要时钟线，数据线，因此比较节省 I/O 口。如下图所示：



利用 74LS164 级联驱动 8 个单独的数码管

静态显示的优点是程序编写简单。但是由于涉及到的驱动芯片数量比较多，同时考虑到 PCB 的布线等等因素，在低成本要求的开发环境下，单纯的静态驱动并不合适。这个时候就可以考虑到动态驱动了。

动态驱动的图如下所示(以 EE21 开发板为例)



由上图可以看出。8 个数码管的段码由一个单独的 74HC573 驱动。同时每一个数码管的公共端连接在另外一个 74HC573 的输出上。当送出第一位数码管的段码内容时候，同时选通第一位数码管的位选，此时，第一位数码管就显示出相应的内容了。一段时间之后，送出第二位数码管段码的内容，选通第二位数码管的位选，这时显示的内容就变成第二位数码管的内容了……依次循环下去，就可以看到了所有数码管同时显示了。事实上，任意时刻，只有一位数码管是被点亮的。由于人眼的视觉暂留效应以及数码管的余辉效应，当数码管扫描的频率非常快的时候，人眼已经无法分辨出数码管的变化了，看起来就是同时点亮的。我们假设数码管的扫描频率为 50 Hz，则完成一轮扫描的时间就是 1 / 50 = 20 ms 。我们的系统共有 8 位数码管，则每一位数码管在一轮扫描周期中点亮的时间为 20 / 8 = 2.5 ms 。

动态扫描对时间要求有一点点严格，否则，就会有明显的闪烁。

假设我们程序中所有任务如下：

```
while(1)
{
    LedDisplay() ;    //数码管动态扫描
    ADProcess() ;     //AD 采集处理
    TimerProcess() ;  //时间相关处理
    DataProcess() ;   //数据处理
}
```

LedDisplay() 这个任务的执行时间，如同我们刚才计算的那样，50 Hz 频率扫描，则该函数执行的时间为 20 ms 。假设 ADProcess()这个任务执行的的时间为 2 ms ，TimerProcess() 这个函数执行的时间为 1 ms ，DataProcess() 这个函数执行的时间为 10 ms 。 那么整个主函数执行一遍的总时间为 20 + 2 + 1 + 10 = 33 ms 。即 LedDisplay() 这个函数的扫描频率已经不为 50 Hz 了，而是 1 / 33 = 30.3 Hz 。这个频率数码管已经可以感觉到闪烁了，因此不符合我们的要求。为什么会出现这种情况呢？ 我们刚才计算的 50 Hz 是系统只有 LedDisplay()这一个任务的时候得出来的结果。当系统添加了其它任务后，当然系统循环执行一次的总时间就增加了。如何解决这种现象了，还是离不开我们第二章所讲的那个思想。系统产生一个 2.5 ms 的时标消息。LedDisplay() ，每次接收到这个消息的时候，扫描一位数码管。这样 8 个时标消息过后，所有的数码管就都被扫描一遍了。可能有朋友会有这样的疑问：ADProcess() 以及 DataProcess() 等函数执行的时间还是需要十几 ms 啊，在这十几 ms 的时间里，已经产生好几个 2.5 ms 的时标消息了，这样岂不是漏掉了扫描，显示起来还是会闪烁。能够想到这一点，很不错，这也就是为什么我们要学会释放 CPU 的原因。对于 ADProcess()，TimerProcess()，DataProcess()，等任务我们依旧要采取此方法对 CPU 进行释放，使其执行的时间尽可能短暂，关于如何做到这一点，在以后的讲解如何设计多任务程序设计的时候会讲解到。

下面我们基于此思路开始编写具体的程序。

首先编写 Timer.c 文件。该文件中主要为系统提供时间相关的服务。必要的头文件包含。

```
#include <reg52.h>
#include "MacroAndConst.h"
```

为了方便计算，我们取数码管扫描一位的时间为 2 ms。设置定时器 0 为 2 ms 中断一次。同时声明一个位变量，作为 2 ms 时标消息的标志

```
bit g_bSystemTime2Ms = 0 ;                // 2msLED 动态扫描时标消息
```

初始化定时器 0

```
void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;      //定时器 0 工作方式 1
    TH0  =  0xf8 ;      //定时器初始值
    TL0  =  0xcc ;
    TR0  = 1 ;
    ET0  = 1 ;
}
```

在定时器 0 中断处理程序中，设置时标消息。

```
void Time0Isr(void) interrupt 1
{
    TH0  =    0xf8 ;            //定时器重新赋初值
```

```
    TL0  =  0xcc ;
    g_bSystemTime2Ms = 1 ;     //2MS 时标标志位置位
}
```

然后我们开始编写数码管的动态扫描函数。

新建一个 C 源文件，并包含相应的头文件。

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Timer.h"
```

先开辟一个数码管显示的缓冲区。动态扫描函数负责从这个缓冲区中取出数据，并扫描显示。而其它函数则可以修改该缓冲区，从而改变显示的内容。

```
uint8 g_u8LedDisplayBuffer[8] = {0} ; //显示缓冲区
```

然后定义共阳数码管的段码表以及相应的硬件端口连接。

```
code uint8 g_u8LedDisplayCode[]=
{
    0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,
    0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,
    0xbf, //'-'号代码
} ;

sbit io_led_seg_cs = P1^4 ;
sbit io_led_bit_cs = P1^5 ;

#define LED_PORT P0
```

再分别编写送数码管段码函数，以及位选通函数。

```
static void SendLedSegData(uint8 dat)
{
    LED_PORT       = dat ;
    io_led_seg_cs = 1 ;          //开段码锁存,送段码数据
    io_led_seg_cs = 0 ;
}

static void SendLedBitData(uint8 dat)
{
    uint8 temp ;
    temp = (0x01 << dat ) ;   //根据要选通的位计算出位码
    LED_PORT       = temp ;
    io_led_bit_cs = 1 ;           //开位码锁存,送位码数据
    io_led_bit_cs = 0 ;
}
```

下面的核心就是如何编写动态扫描函数了。

如下所示：

```
void LedDisplay(uint8 * pBuffer)
{
    static uint8 s_LedDisPos = 0 ;
    if(g_bSystemTime2Ms)
    {
        g_bSystemTime2Ms = 0 ;

        SendLedBitData(8) ;          //消隐，只需要设置位选不为 0~7 即可

        if(pBuffer[s_LedDisPos] == '-')       //显示'-'号
        {
            SendLedSegData(g_u8LedDisplayCode[16]) ;
        }
        else
        {
            SendLedSegData(g_u8LedDisplayCode[pBuffer[s_LedDisPos]]) ;
        }

        SendLedBitData(s_LedDisPos);

        if(++s_LedDisPos > 7)
        {
            s_LedDisPos = 0 ;
        }
    }
}
```

关闭数码管位

函数内部定义一个静态的变量 s_LedDisPos，用来表示扫描数码管的位置。每当我们执行该函数一次的时候，s_LedDisPos 的值会自加 1，表示下次扫描下一个数码管。然后判断 g_bSystemTime2Ms 时标消息是否到了。如果到了，就开始执行相关扫描，否则就直接跳出函数。SendLedBitData(8) ;的作用是消隐。因为我们的系统的段选和位选是共用 P0 口的。在送段码之前，必须先关掉位选，否则，因为上次位选是选通的，在送段码的时候会造成相应数码管的点亮，尽管这个时间很短暂。但是因为我们的数码管是不断扫描的，所以看起来还是会有些微微亮。为了消除这种影响，就有必要再送段码数据之前关掉位选。

if(pBuffer[s_LedDisPos] == '-')　　//显示'-'号这行语句是为了显示'-'符号特意加上去的，大家可以看到在定义数码管的段码表的时候，我多加了一个字节的代码 0xbf：

```
code uint8 g_u8LedDisplayCode[]=
{
    0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,
    0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,
    0xbf, //'-'号代码
} ;
```

通过 SendLedSegData(g_u8LedDisplayCode[pBuffer[s_LedDisPos]]) ;送出相应的段码数

据后，然后通过 SendLedBitData(s_LedDisPos);打开相应的位选。这样对应的数码管就被点亮了。

```
if(++s_LedDisPos > 7)
{
    s_LedDisPos = 0 ;
}
```

然后 s_LedDisPos 自加 1，以便下次执行本函数时，扫描下一个数码管。因为我们的系统共有 8 个数码管，所以当 s_LedDisPos > 7 后，要对其进行清 0 。否则，没有任何一个数码管被选中。这也是为什么我们可以用

SendLedBitData(8) ;　　//消隐，只需要设置位选不为 0~7 即可对数码管进行消隐操作的原因。

下面我们来编写相应的主函数，并实现数码管上面类似时钟的效果，如显示 10-20-30 即 10 点 20 分 30 秒。

Main.c

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Timer.h"
#include "Led7Seg.h"

sbit io_led = P1^6 ;


void main(void)
{
        io_led = 0 ;      //发光二极管与数码管共用 P0 口,这里禁止掉发光二极管的锁存输出
        Timer0Init() ;
        g_u8LedDisplayBuffer[0] = 1 ;
        g_u8LedDisplayBuffer[1] = 0 ;
        g_u8LedDisplayBuffer[2] = '-' ;
        g_u8LedDisplayBuffer[3] = 2 ;
        g_u8LedDisplayBuffer[4] = 0 ;
        g_u8LedDisplayBuffer[5] = '-' ;
        g_u8LedDisplayBuffer[6] = 3 ;
        g_u8LedDisplayBuffer[7] = 0 ;
        EA = 1 ;
        while(1)
        {
            LedDisplay(g_u8LedDisplayBuffer) ;
        }
}
```

将整个工程进行编译，看看效果如何

动起来

既然我们想要模拟一个时钟，那么时钟肯定是要走动的，不然还称为什么时钟撒。下面我们在前面的基础之上，添加一点相应的代码，让我们这个时钟走动起来。
我们知道，之前我们以及设置了一个扫描数码管用到的 2 ms 时标。 如果我们再对这个时标进行计数，当计数值达到 500，即 500 * 2 = 1000 ms 时候，即表示已经逝去了 1 S 的时间。我们再根据这个 1 S 的时间更新显示缓冲区即可。听起来很简单，让我们实现它吧。
首先在 Timer.c 中声明如下两个变量：

```
bit g_bTime1S = 0 ;                          //时钟 1S 时标消息
static uint16 s_u16ClockTickCount = 0 ;  //对 2 ms 时标进行计数
```

再在定时器中断函数中添加如下代码：

```
    if(++s_u16ClockTickCount == 500)
    {
        s_u16ClockTickCount = 0 ;
        g_bTime1S = 1 ;
    }
```

从上面可以看出，s_u16ClockTickCount 计数值达到 500 的时候，g_bTime1S 时标消息产生。然后我们根据这个时标消息刷新数码管显示缓冲区：

```
void RunClock(void)
{
    if(g_bTime1S )
    {
        g_bTime1S = 0 ;
        if(++g_u8LedDisplayBuffer[7] == 10)
        {
            g_u8LedDisplayBuffer[7] = 0 ;
```

```
            if(++g_u8LedDisplayBuffer[6] == 6)
            {
                g_u8LedDisplayBuffer[6] = 0 ;
                if(++g_u8LedDisplayBuffer[4] == 10)
                {
                    g_u8LedDisplayBuffer[4]     = 0 ;
                    if(++g_u8LedDisplayBuffer[3] == 6)
                    {
                        g_u8LedDisplayBuffer[3] = 0 ;
                        if(     g_u8LedDisplayBuffer[0]<2)
                        {
                            if(++g_u8LedDisplayBuffer[1]==10)
                            {
                                g_u8LedDisplayBuffer[1] = 0 ;
                                g_u8LedDisplayBuffer[0]++;
                            }
                        }
                        else
                        {
                            if(++g_u8LedDisplayBuffer[1]==4)
                            {
                                g_u8LedDisplayBuffer[1] = 0 ;
                                g_u8LedDisplayBuffer[0] = 0 ;
                            }
                        }
                    }
                }
            }

        }
    }
}
```

这个函数的作用就是对每个数码管缓冲位的值进行判断，判断的标准就是我们熟知的 24 小时制。如秒的个位到了 10 就清 0，同时秒的十位加 1….诸如此类，我就不一一详述了。同时，我们再编写一个时钟初始值设置函数，这样，可以很方便的在主程序开始的时候修改时钟初始值。

```
void SetClock(uint8 nHour, uint8 nMinute, uint8 nSecond)
{
    g_u8LedDisplayBuffer[0] = nHour / 10 ;
    g_u8LedDisplayBuffer[1] = nHour % 10 ;
    g_u8LedDisplayBuffer[2] = '-' ;
    g_u8LedDisplayBuffer[3] = nMinute / 10 ;
    g_u8LedDisplayBuffer[4] = nMinute % 10 ;
    g_u8LedDisplayBuffer[5] = '-' ;
```

```
    g_u8LedDisplayBuffer[6] = nSecond / 10 ;
    g_u8LedDisplayBuffer[7] = nSecond % 10 ;
}
```

然后修改下我们的主函数如下：

```
void main(void)
{
    io_led = 0 ;       //发光二极管与数码管共用 P0 口,这里禁止掉发光二极管的锁存输出
    Timer0Init() ;
    SetClock(10,20,30) ;  //设置初始时间为 10 点 20 分 30 秒
    EA = 1 ;
    while(1)
    {
        LedDisplay(g_u8LedDisplayBuffer) ;
        RunClock();
    }
}
```

编译好之后，下载到我们的实验板上，怎么样，一个简单的时钟就这样诞生了。

至此，本章所诉就告一段落了。至于如何完成数码管的闪烁显示，就像本章开头所说的那个数码管时钟的功能，就作为一个思考的问题留给大家思考吧。

同时整个 LED 篇就到此结束了，在以后的文章中，我们将开始学习如何编写实用的按键扫描程序。

[/post

本章所附例程在 EE21 学习板上调试通过，拥有板子的朋友可以直接下载附件对照学习

六、 KEY 主题讨论第一章——按键程序编写的基础

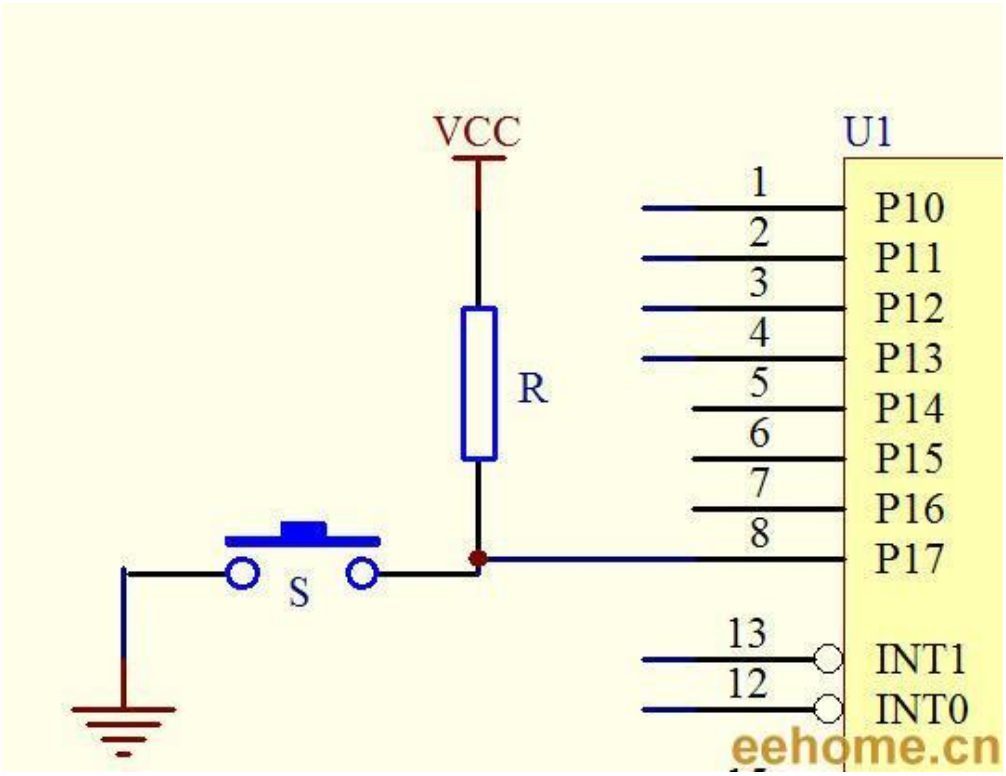
从这一章开始，我们步入按键程序设计的殿堂。在基于单片机为核心构成的应用系统中，用户输入是必不可少的一部分。输入可以分很多种情况，譬如有的系统支持 PS2 键盘的接口，有的系统输入是基于编码器，有的系统输入是基于串口或者 USB 或者其它输入通道等等。在各种输入途径中，更常见的是，基于单个按键或者由单个键盘按照一定排列构成的矩阵键盘(行列键盘)。我们这一篇章主要讨论的对象就是基于单个按键的程序设计，以及矩阵键盘的程序编写。

◎按键检测的原理

常见的独立按键的外观如下，相信大家并不陌生，各种常见的开发板学习板上随处可以看到他们的身影。

总共有四个引脚，一般情况下，处于同一边的两个引脚内部是连接在一起的，如何分辨两个引脚是否处在同一边呢？可以将按键翻转过来，处于同一边的两个引脚，有一条突起的线将他们连接一起，以标示它们俩是相连的。如果无法观察得到，用数字万用表的二极管挡位检测一下即可。搞清楚这点非常重要，对于我们画 PCB 的时候的封装很有益。

它们和我们的单片机系统的 I/O 口连接一般如下：

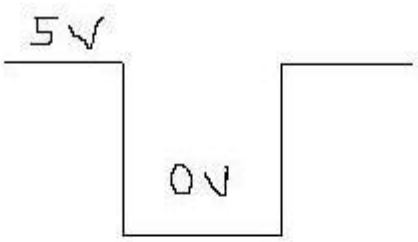


对于单片机 I/O 内部有上拉电阻的微控制器而言，还可以省掉外部的那个上拉电阻。简

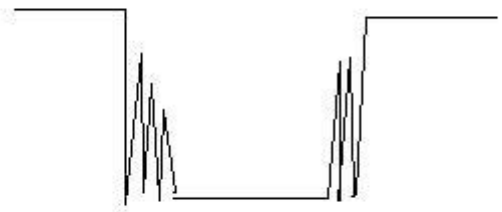
单分析一下按键检测的原理。当按键没有按下的时候，单片机 I/O 通过上拉电阻 R 接到 VCC，我们在程序中读取该 I/O 的电平的时候，其值为 1(高电平)；当按键 S 按下的时候，该 I/O 被短接到 GND，在程序中读取该 I/O 的电平的时候，其值为 0(低电平) 。这样，按键的按下与否，就和与该按键相连的 I/O 的电平的变化相对应起来。结论：我们在程序中通过检测到该 I/O 口电平的变化与否，即可以知道按键是否被按下，从而做出相应的响应。一切看起来很美好，是这样的吗？

◎现实并非理想

在我们通过上面的按键检测原理得出上述的结论的时候，其实忽略了一个重要的问题，那就是现实中按键按下时候的电平变化状态。我们的结论是基于理想的情况得出来的，就如同下面这幅按键按下时候对应电平变化的波形图一样：



而实际中，由于按键的弹片接触的时候，并不是一接触就紧紧的闭合，它还存在一定的抖动，尽管这个时间非常的短暂，但是对于我们执行时间以 us 为计算单位的微控制器来说，它太漫长了。因而，实际的波形图应该如下面这幅示意图一样。



这样便存在这样一个问题。假设我们的系统有这样功能需求：在检测到按键按下的时候，将某个 I/O 的状态取反。由于这种抖动的存在，使得我们的微控制器误以为是多次按键的按下，从而将某个 I/O 的状态不断取反，这并不是我们想要的效果，假如该 I/O 控制着系统中某个重要的执行的部件，那结果更不是我们所期待的。于是乎有人便提出了软件消除抖动的思想，道理很简单：抖动的时间长度是一定的，只要我们避开这段抖动时期，检测稳定的时候的电

平不久可以了吗？听起来确实不错，而且实际应用起来效果也还可以。于是，各种各样的书籍中，在提到按键检测的时候，总也不忘说道软件消抖。就像下面的伪代码所描述的一样。(假设按键按下时候，低电平有效)

```
If(0 == io_KeyEnter)              //如果有键按下了
{
   Delayms(20) ;                  //先延时 20ms 避开抖动时期
    If(0 == io_KeyEnter)          //然后再检测，如果还是检测到有键按下
    {
        return KeyValue ;         //是真的按下了，返回键值
    }
    else
{
    return KEY_NULL               //是抖动，返回空的键值
}
    while(0 == io_KeyEnter) ;     //等待按键释放
}
```

乍看上去，确实挺不错，实际中呢？在实际的系统中，一般是不允许这么样做的。为什么呢？首先，这里的 Delayms(20)，让微控制器在这里白白等待了 20 ms 的时间，啥也没干，考虑我在《学会释放 CPU》一章中所提及的几点，这是不可取的。其次 while(0 == io_KeyEnter);更是程序设计中的大忌(极少的特殊情况例外)。所以合理的分配好微控制的处理时间，是编写按键程序的基础。任何非极端情况下，都不要使用这样语句来堵塞微控制器的执行进程。原本是等待按键释放，结果 CPU 就一直死死的盯住该按键，其它事情都不管了，那其它事情不干了吗？你同意别人可不会同意

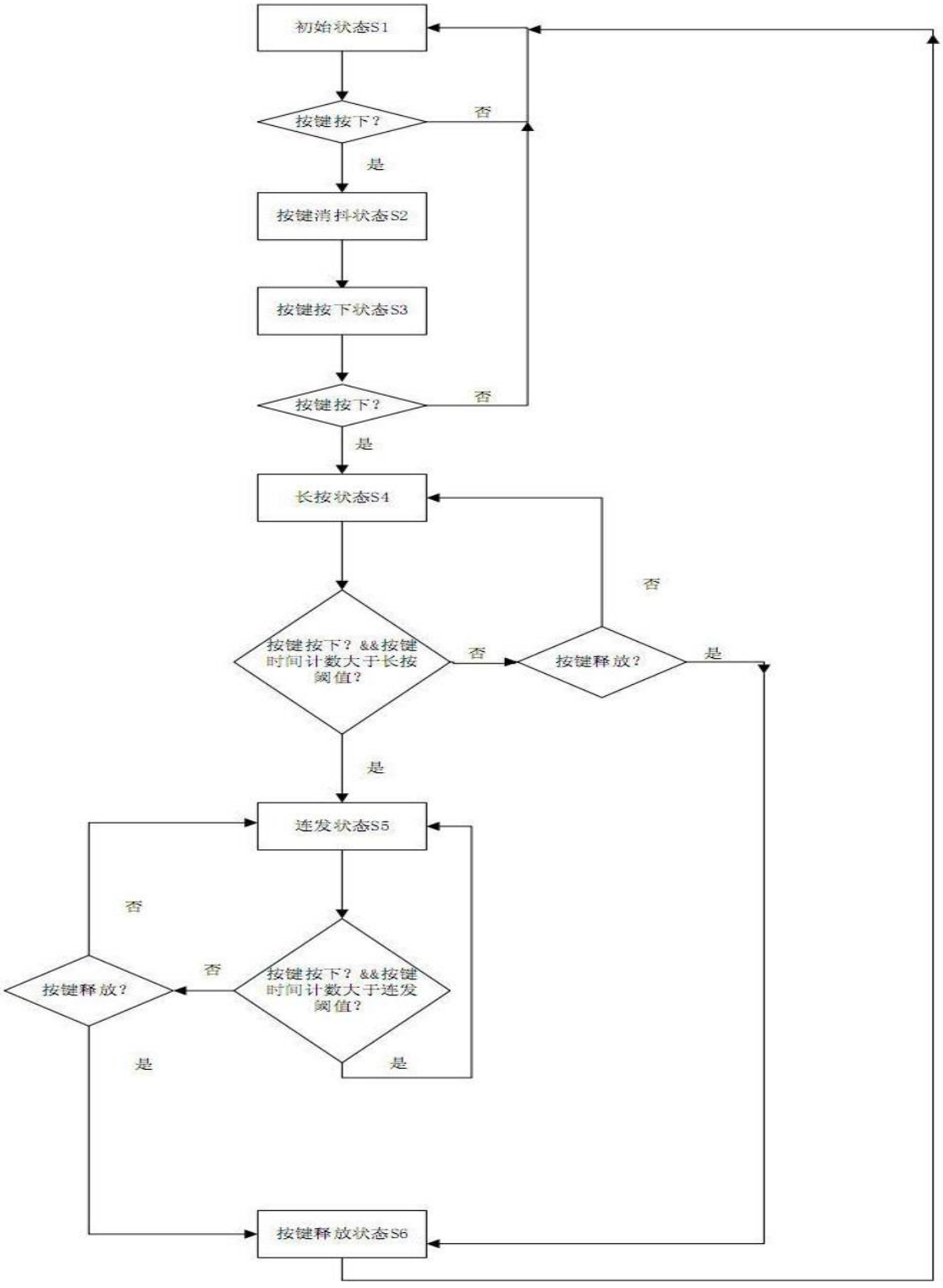
◎消除抖动有必要吗？

的确，软件上的消抖确实可以保证按键的有效检测。但是，这种消抖确实有必要吗？有人提出了这样的疑问。抖动是按键按下的过程中产生的，如果按键没有按下，抖动会产生吗？如果没有按键按下，抖动也会在 I/O 上出现，我会立刻把这个微控制器锤了，永远不用这样一款微控制器。所以抖动的出现即意味着按键已经按下，尽管这个电平还没有稳定。所以只要我们检测到按键按下，即可以返回键值，问题的关键是，在你执行完其它任务的时候，再次执行我们的按键任务的时候，抖动过程还没有结束，这样便有可能造成重复检测。所以，如何在返回键值后，避免重复检测，或者在按键一按下就执行功能函数，当功能函数的执行时间小于抖动时间时候，如何避免再次执行功能函数，就成为我们要考虑的问题了。这是一个仁者见仁，智者见智的问题，就留给大家去思考吧。所以消除抖动的目的是：防止按键一次按下，多次响应。

七、 KEY 主题讨论第二章——基于状态转移的独立按键程序设计

本章所描述的按键程序要达到的目的：检测按键按下，短按，长按，释放。即通过按键的返回值我们可以获取到如下的信息：按键按下(短按)，按键长按，按键连发，按键释放。不知道大家还记得小时候玩过的电子钟没有，就是外形类似于 CALL 机(CALL)的那种，有一个小液晶屏，还有四个按键，功能是时钟，闹钟以及秒表。在调整时间的时候，短按+键每次调整值加一，长按的时候调整值连续增加。小的时候很好奇，这样的功能到底是如何实现的呢，今天就让我们来剖析它的原理吧。状态机，好像是很古老的东西了

状态在生活中随处可见。譬如早上的时候，闹钟把你叫醒了，这个时候，你便处于清醒的状态，马上你就穿衣起床洗漱吃早餐，这一系列事情就是你在这个状态做的事情。做完这些后你会去等车或者开车去上班，这个时候你就处在上班途中的状态.....中午下班时间到了，你就处于中午下班的状态，诸如此类等等，在每一个状态我们都会做一些不同的事情，而总会有外界条件促使我们转换到另外一种状态，譬如闹钟叫醒我们了，下班时间到了等等。对于状态的定义出发点不同，考虑的方向不同，或者会有些许细节上面的差异，但是大的状态总是相同的。生活中的事物同样遵循同样的规律，譬如，用一个智能充电器给你的手机电池充电，刚开始，它是处于快速充电状态，随着电量的增加，电压的升高，当达到规定的电压时候，它会转换到恒压充电。总而言之，细心观察，你会发现生活中的总总都可以归结为一个个的状态，而状态的变换或者转移总是由某些条件引起同时伴随着一些动作的发生。我们的按键亦遵循同样的规律，下面让我们来简单的描绘一下它的状态流程转移图。



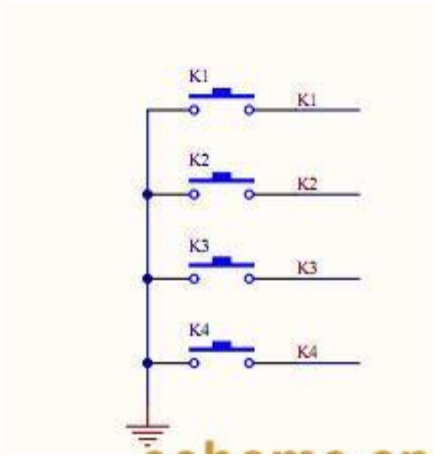
下面对上面的流程图进行简要的分析。

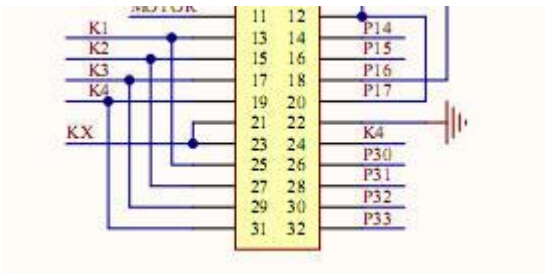
首先按键程序进入初始状态 S1，在这个状态下，检测按键是否按下，如果有按下，则进入按键消抖状态 2，在下一次执行按键程序时候，直接由按键消抖状态进入按键按下状态 3，在此状态下检测按键是否按下，如果没有按键按下，则返回初始状态 S1，如果有则可以返回键值，同时进入长按状态 S4，在长按状态下每次进入按键程序时候对按键时间计数，当计数值超过设定阈值时候，则表明长按事件发生，同时进入按键连发状态 S5。如果按键键

值为空键，则返回按键释放状态 S6，否则继续停留在本状态。在按键连发状态下，如果按键键值为空键则返回按键释放状态 S6，如果按键时间计数超过连发阈值，则返回连发按键值，清零时间计数后继续停留在本状态。

看了这么多，也许你已经有一个模糊的概念了，下面让我们趁热打铁，一起来动手编写按键驱动程序吧。

下面是我使用的硬件的连接图。





硬件连接很简单，四个独立按键分别接在 P3^0------P3^3 四个 I/O 上面。

因为 51 单片机 I/O 口内部结构的限制，在读取外部引脚状态的时候，需要向端口写 1。在 51 单片机复位后，不需要进行此操作也可以进行读取外部引脚的操作。因此，在按键的端口没有复用的情况下，可以省略此步骤。而对于其它一些真正双向 I/O 口的单片机来说，将引脚设置成输入状态，是必不可少的一个步骤。

下面的程序代码初始化引脚为输入:

```
void KeyInit(void)
{
    io_key_1 = 1 ;
    io_key_2 = 1 ;
    io_key_3 = 1 ;
    io_key_4 = 1 ;
}
```

根据按键硬件连接定义按键键值

```
#define KEY_VALUE_1          0x0e
#define KEY_VALUE_2          0x0d
#define KEY_VALUE_3          0x0b
#define KEY_VALUE_4          0x07
```

```
#define KEY_NULL                    0x0f
```
下面我们来编写按键的硬件驱动程序。
根据第一章所描述的按键检测原理，我们可以很容易的得出如下的代码：
```
static uint8 KeyScan(void)
{
    if(io_key_1 == 0)return KEY_VALUE_1 ;
    if(io_key_2 == 0)return KEY_VALUE_2 ;
    if(io_key_3 == 0)return KEY_VALUE_3 ;
    if(io_key_4 == 0)return KEY_VALUE_4 ;
    return KEY_NULL ;
}
```
其中 io_key_1 等是我们按键端口的定义，如下所示：
```
sbit io_key_1 = P3^0 ;
sbit io_key_2 = P3^1 ;
sbit io_key_3 = P3^2 ;
sbit io_key_4 = P3^3 ;
```

KeyScan()作为底层按键的驱动程序，为上层按键扫描提供一个接口，这样我们编写的上层按键扫描函数可以几乎不用修改就可以拿到我们的其它程序中去使用，使得程序复用性大大提高。同时，通过有意识的将与底层硬件连接紧密的程序和与硬件无关的代码分开写，使得程序结构层次清晰，可移植性也更好。对于单片机类的程序而言，能够做到函数级别的代码重用已经足够了。
在编写我们的上层按键扫描函数之前，需要先完成一些宏定义。
```
//定义长按键的 TICK 数,以及连发间隔的 TICK 数
#define KEY_LONG_PERIOD         100
#define KEY_CONTINUE_PERIOD     25

//定义按键返回值状态(按下,长按,连发,释放)
#define KEY_DOWN                0x80
#define KEY_LONG                0x40
#define KEY_CONTINUE            0x20
#define KEY_UP                  0x10

//定义按键状态
#define KEY_STATE_INIT          0
#define KEY_STATE_WOBBLE        1
#define KEY_STATE_PRESS         2
#define KEY_STATE_LONG          3
#define KEY_STATE_CONTINUE      4
#define KEY_STATE_RELEASE       5
```

接着我们开始编写完整的上层按键扫描函数，按键的短按，长按，连按，释放等等状态的判断均是在此函数中完成。对照状态流程转移图，然后再看下面的函数代码，可以更容易的去理解函数的执行流程。完整的函数代码如下：

```
void GetKey(uint8 *pKeyValue)
{
    static uint8 s_u8KeyState = KEY_STATE_INIT ;
    static uint8 s_u8KeyTimeCount = 0 ;
    static uint8 s_u8LastKey = KEY_NULL ;  //保存按键释放时候的键值
    uint8 KeyTemp = KEY_NULL ;

    KeyTemp = KeyScan() ;          //获取键值

    switch(s_u8KeyState)
    {
        case KEY_STATE_INIT :
                {
                    if(KEY_NULL != (KeyTemp))
                    {
                        s_u8KeyState = KEY_STATE_WOBBLE ;
                    }
                }
        break ;

        case KEY_STATE_WOBBLE :       //消抖
                {
                    s_u8KeyState = KEY_STATE_PRESS ;
                }
        break ;

        case KEY_STATE_PRESS :
                {
                    if(KEY_NULL != (KeyTemp))
                    {
                        s_u8LastKey = KeyTemp ; //保存键值,以便在释放按键状态返回键值
                        KeyTemp |= KEY_DOWN ;  //按键按下
                        s_u8KeyState = KEY_STATE_LONG ;
                    }
                    else
                    {
                        s_u8KeyState = KEY_STATE_INIT ;
                    }
                }
        break ;

        case KEY_STATE_LONG :
                {
```

```
                if(KEY_NULL != (KeyTemp))
                {
                    if(++s_u8KeyTimeCount > KEY_LONG_PERIOD)
                    {
                        s_u8KeyTimeCount = 0 ;
                        KeyTemp |= KEY_LONG ;  //长按键事件发生
                        s_u8KeyState = KEY_STATE_CONTINUE ;
                    }
                }
                else
                {
                    s_u8KeyState = KEY_STATE_RELEASE ;
                }
            }
        break ;

        case KEY_STATE_CONTINUE :
            {
                if(KEY_NULL != (KeyTemp))
                {
                    if(++s_u8KeyTimeCount > KEY_CONTINUE_PERIOD)
                    {
                        s_u8KeyTimeCount = 0 ;
                        KeyTemp |= KEY_CONTINUE ;
                    }
                }
                else
                {
                    s_u8KeyState = KEY_STATE_RELEASE ;
                }
            }
        break ;
        case KEY_STATE_RELEASE :
            {
                s_u8LastKey |= KEY_UP ;
                KeyTemp = s_u8LastKey ;
                s_u8KeyState = KEY_STATE_INIT ;
            }
        break ;

        default : break ;
    }
    *pKeyValue = KeyTemp ; //返回键值
}
```

关于这个函数内部的细节我并不打算花过多笔墨去讲解。对照着按键状态流程转移图，然后去看程序代码，你会发现其实思路非常清晰。最能让人理解透彻的，莫非就是将整个程序自己看懂，然后想象为什么这个地方要这样写，抱着思考的态度去阅读程序，你会发现自己的程序水平会慢慢的提高。所以我更希望的是你能够认认真真的看完，然后思考。也许你会收获更多。

不管怎么样，这样的一个程序已经完成了本章开始时候要求的功能：按下，长按，连按，释放。事实上，如果掌握了这种基于状态转移的思想，你会发现要求实现其它按键功能，譬如，多键按下，功能键等等，亦相当简单，在下一章，我们就去实现它。

在主程序中我编写了这样的一段代码，来演示我实现的按键功能。

```
void main(void)
{
    uint8 KeyValue = KEY_NULL;
    uint8 temp = 0 ;
      LED_CS11 = 1 ; //流水灯输出允许
    LED_SEG = 0 ;
    LED_DIG = 0 ;
    Timer0Init() ;
    KeyInit() ;
    EA = 1 ;
    while(1)
    {
        Timer0MainLoop() ;
        KeyMainLoop(&KeyValue) ;

        if(KeyValue == (KEY_VALUE_1 | KEY_DOWN)) P0 = ~1 ;
        if(KeyValue == (KEY_VALUE_1 | KEY_LONG)) P0 = ~2 ;
        if(KeyValue == (KEY_VALUE_1 | KEY_CONTINUE)) { P0 ^= 0xf0;}
        if(KeyValue == (KEY_VALUE_1 | KEY_UP)) P0 = 0xa5 ;
    }

}
```

按住第一个键，可以清晰的看到 P0 口所接的 LED 的状态的变化。当按键按下时候，第一个 LED 灯亮，等待 2 S 后第二个 LED 亮，第一个熄灭，表示长按事件发生。再过 500 ms 第 5~8 个 LED 闪烁，表示连按事件发生。当释放按键时候，P0 口所接的 LED 的状态为：灭亮灭亮亮灭亮灭，这也正是 P0 = 0xa5 这条语句的功能

再给大家分享一个不错按键程序（来自 ourdev）

```
/****************************************
键盘_不采用定时器_不延时

特点：
按键在松手后有效,灵敏度高,消耗资源少,运行效率高

独立键盘为:K01=P2^4;K02=P2^5;K03=P2^6;K04=P2^7;
矩阵键盘为:行(上到下)_P2.3_P2.2_P2.1_P2.0
           列(左到右)_P2.7_P2.6_P2.5_P2.4

提供的操作函数：

//独立键盘.无按键动作时其返回值 num_key=0,否则返回按键号 num_key
extern unsigned char keyboard_self();

//矩阵键盘.无按键动作时其返回值 num_key=0,否则返回按键号 num_key****检测高四位
extern unsigned char keyboard_matrix();

****************************************/

先看独立键盘（和矩阵键盘的算法一样）
----------------------------------------------------------------------
#include<reg52.h>
#include<intrins.h>

//独立键盘.无按键动作时其返回值 num_key=0,否则返回按键号 num_key
extern unsigned char keyboard_self()
{
    unsigned char num_key=0;//按键号
    unsigned char temp=0;//用于读取 P2 线上按键值
    static unsigned char temp_code=0;//保存按键值
    static unsigned char num_check=0;//低电平有效次数
    static unsigned char key_flag=0;//按键有效标识

    temp=P2&0xF0;//读取 P2 线数据

    if(temp!=0xF0)//低电平判断
    {
        num_check++;
        if(num_check==10)//连续 10 次(10ms)低电平有效,则认为按键有效
        {
            key_flag=1;//使能按键有效标识
            temp_code=temp;//保存按键值
```

```
        }
    }
    else//松手时判断
    {
        num_check=0;

        if(key_flag==1)//按键有效
        {
            key_flag=0;

            switch(temp_code)//读取按键号
            {
                case 0xE0: num_key=1;
                        break;
                case 0xD0: num_key=2;
                        break;
                case 0xB0: num_key=3;
                        break;
                case 0x70: num_key=4;
                        break;
            }
        }
    }

    return(num_key);
}
```

现在是矩阵键盘的

```
#include<reg52.h>
#include<intrins.h>

//矩阵键盘.无按键动作时其返回值 num_key=0,否则返回按键号 num_key****检测高四位
extern unsigned char keyboard_matrix()
{
    unsigned char num_key=0;//按键号
    unsigned char temp=0;//读取 P2 口线数据
    static unsigned char temp_code=0;//用于保存按键值
    static unsigned char temp_circle=0xFE;//保存 P2 线上的循环扫描值
    static unsigned char num_check=0;//低电平计数
    static unsigned char key_flag=0;//按键有效标识

    P2=temp_circle;//0xFX
```

```
temp=P2;//读取 P2 口线数据
if(temp!=temp_circle)//有按键动作
{
    num_check++;//低电平计数|逢低电平加 1
    if(num_check==10)//连续 10 次(10ms)低电平有效
    {
        key_flag=1;//按键有效标识置 1
        temp_code=temp;//保存按键值
    }
}
else//松手 OR 无按键动作,此时应该改变扫描线
{
    num_check=0;
    if(key_flag==1)//按键有效判断
    {
        key_flag=0;
        switch(temp_code)//读取按键号
        {
            //P2^0 线
            case 0xEE: num_key=1;
                    break;
            case 0xDE: num_key=2;
                    break;
            case 0xBE: num_key=3;
                    break;
            case 0x7E: num_key=4;
                    break;
            //P2^1 线
            case 0xED: num_key=5;
                    break;
            case 0xDD: num_key=6;
                    break;
            case 0xBD: num_key=7;
                    break;
            case 0x7D: num_key=8;
                    break;
            //P2^2 线
            case 0xEB: num_key=9;
                    break;
            case 0xDB: num_key=10;
                    break;
            case 0xBB: num_key=11;
                    break;
            case 0x7B: num_key=12;
```

```
                    break;
                //P2^3 线
                case 0xE7: num_key=13;
                    break;
                case 0xD7: num_key=14;
                    break;
                case 0xB7: num_key=15;
                    break;
                case 0x77: num_key=16;
                    break;
            }
        }
        temp_circle=_crol_(temp_circle,1);//改变扫描线
        if(temp_circle==0xEF)
        {
            temp_circle=0xFE;
        }
    }
    return(num_key);//返回按键号
}

/*************************************************************************

未按键时,扫描线一直变化。
长按键时,扫描线不变化,使得该行按键变成了独立按键,这样的扫描效率极高。
如当按下 P2.0 线上的某个键时,程序将扫描到这个键,而后扫描线不变化,
当键盘程序连续 10 次进入时检测到 10 次按键有效,直到松手后扫描线才变化

*************************************************************************/
```

八、 综合应用之一——如何设计复杂的多任务程序

我们在入门阶段，一般面对的设计都是单一的简单的任务，流程图可以如图 1 所示，通常会用踏步循环延时来满足任务需要。

面对多任务，稍微复杂的程序设计，沿用图 1 的思想，我们会做出如图 2 所示的程序，在大循环体中不断增加任务，通常还要用延时来满足特定任务节拍，这种程序设计思想它有明显的不足，主要是各个任务之间相互影响，增加新的任何之后，以前很好的运行的任务有可能不正常，例如数码管动态扫描，本来显示效果很好的驱动函数，在增加新的任务后出现闪烁，显示效果变差了。

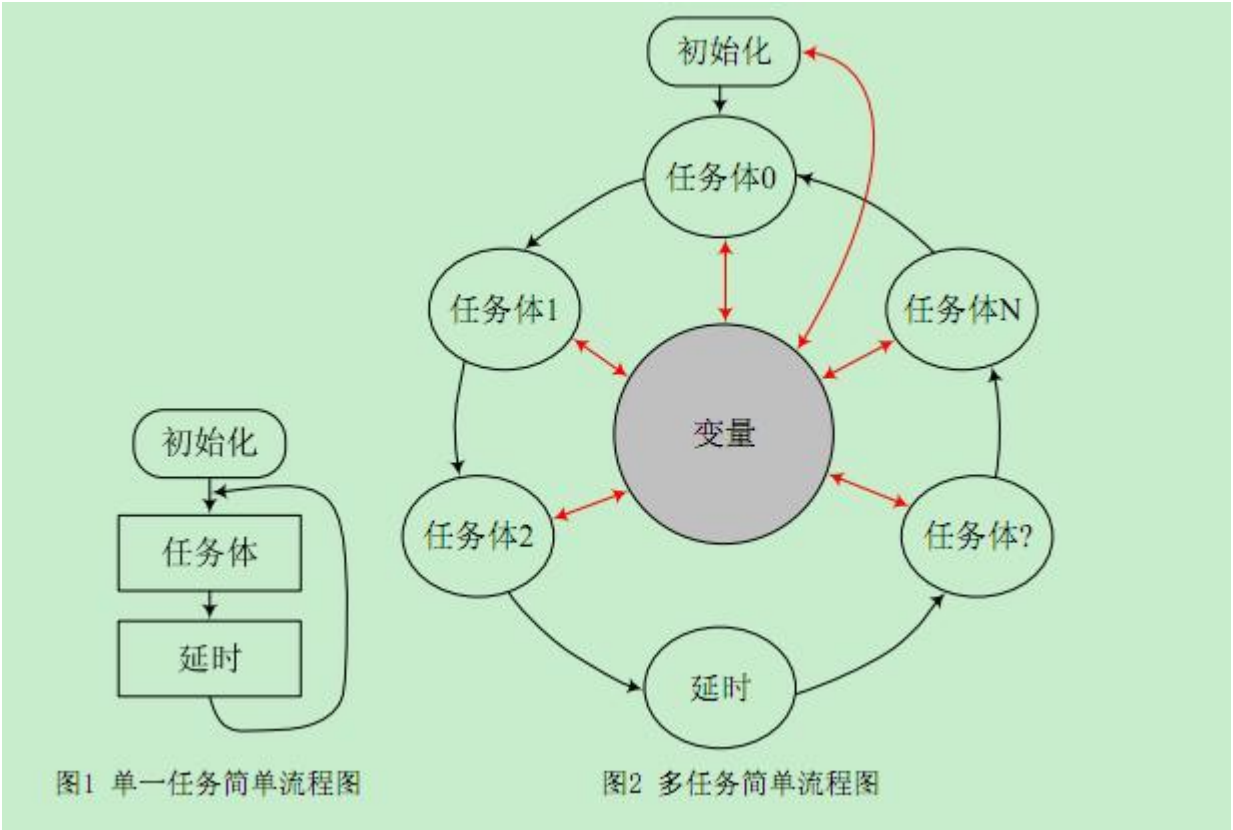


图 1 单一任务简单流程图　　图 2 多任务简单流程图

很明显，初学者在设计程序时，需要从程序构架思想上下功夫，在做了大量基本模块练习之后，需要总结提炼自己的程序设计思路（程序架构思想）。

首先我们来理解“任务”，所谓任务，就是需要 CPU 周期“关照”的事件，绝大多数任务不需要 CPU 一直“关照”，例如启动 ADC 的启动读取。甚至有些任务“害怕”CPU 一直“关照”例如 LCD 的刷新，因为 LCD 是显示给人看的，并不需要高速刷新，即便是显示的内容在高速变化，也不需要高速刷新，道理是一样的。这样看来，让 CPU 做简单任务一定很浪费，事实也是如此，绝大多数简单任务，CPU 都是在“空转”（循环踏步延时）。对任务总结还可以知道，很多任务需要 CPU 不断“关照”，其实这种“不断”也是有极限的，比如数码管动态扫描，能够做到 40Hz 就可以了，又如键盘扫描，能够做到 20Hz（经验值），基本上也就不会丢有效按键键值了，再如 LCD 刷新，我觉得做到 10Hz 就可以了，等等。看

来，绝大多数任务都是工作在低速频度。而我们的 CPU 一旦运行起来，速度又很快，CPU 本身就是靠很快的速度执行很简单的指令来胜任复杂的任务（逻辑）的。如果有办法把“快”的 CPU 分成多个慢的 CPU，然后给不同的任务分配不同速度的 CPU，这种设想是不是很好呢！确实很好，下面就看如何将“快”的 CPU 划分成多个“慢”的 CPU。

根据这种想法，我们需要合理分配 CPU 资源来“关照”不同的任务，最好能够根据任务本身合理占用 CPU 资源，首先看如图 3 所示的流程图，各个任务流程独立，各任务通过全局变量来交互信息，在流程中有一个重要的模块“任务切换”，就是任务切换模块实现 CPU 合理分配，这个任务切换模块是怎么实现的呢？

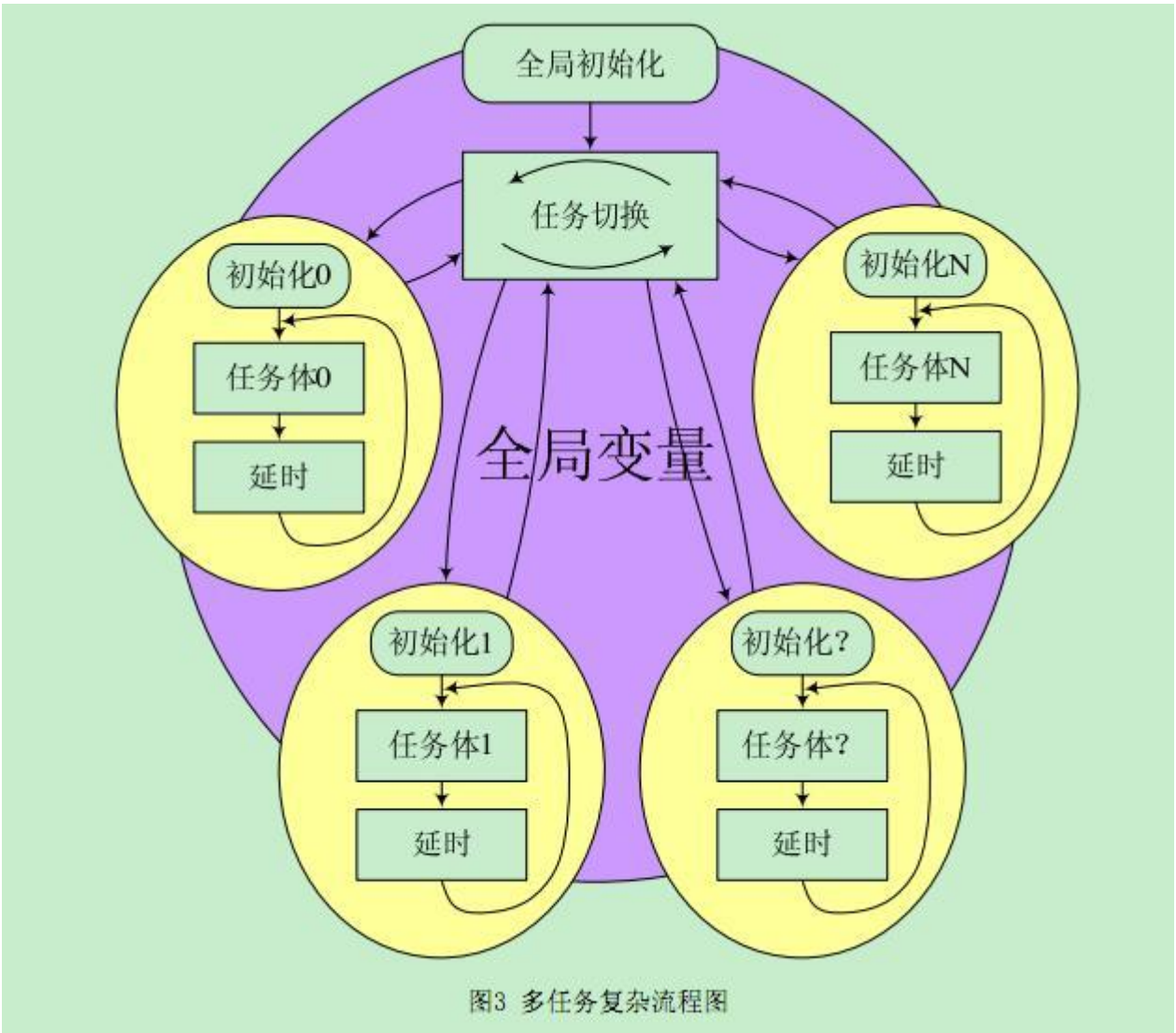


图 3 多任务复杂流程图

首先需要理解，CPU 一旦运行起来，就无法停止（硬件支持时钟停止的不在这里讨论），谁能够控制一批脱缰的马呢？对了，有中断，中断能够让 CPU 回到特定的位置，设想，能不能用一个定时中断，周期性的将 CPU 这匹运行着的脱缰的马召唤回来，重新给它安排特定的 任务，事实上，任务切换就是这样实现的。

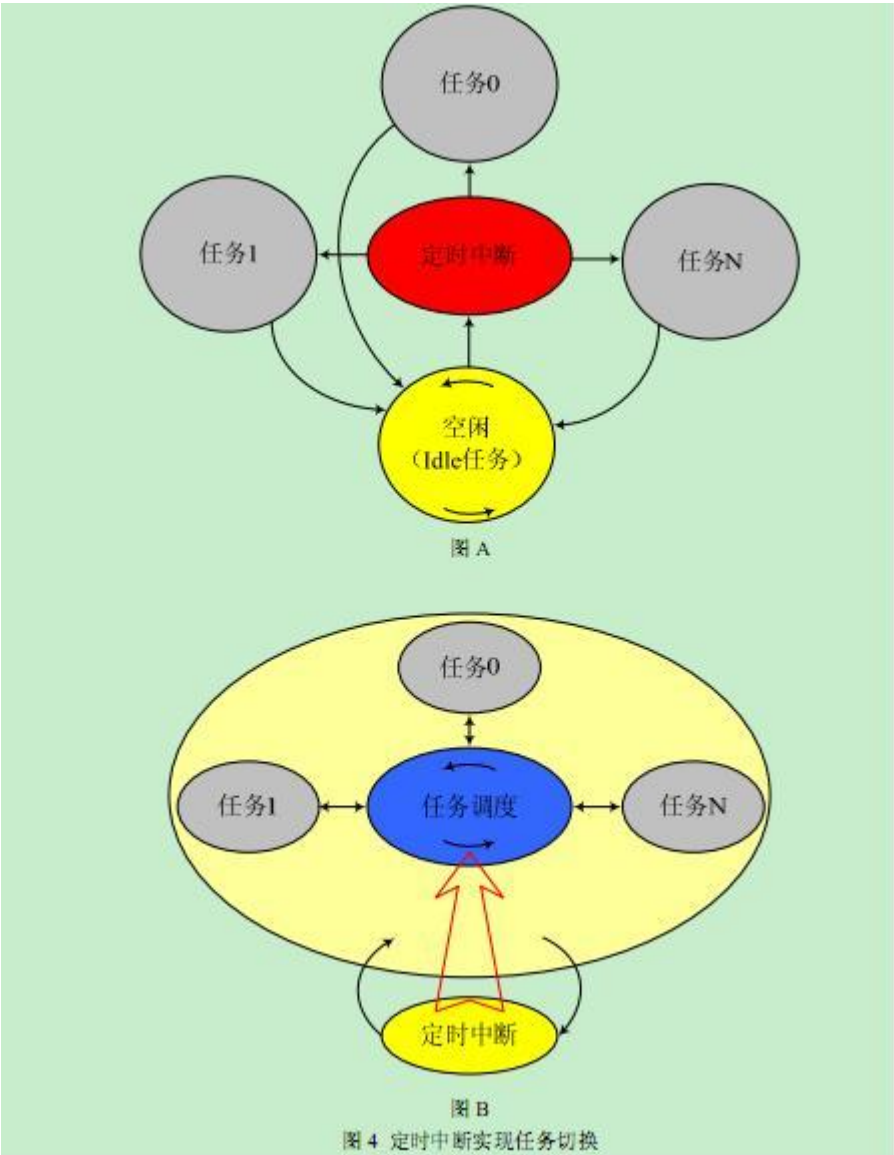


图 4 定时中断实现任务切换

如图 4A 所示，CPU 在空闲任务循环等待，定时中断将 CPU 周期性唤回，根据任务设计了不同的响应频度，满足条件的任务将获得 CPU 资源，CPU 为不同任务“关照”完成后，再次返回空闲任务，如此周而复始，对于各个任务而言，好像各自拥有一个独立的 CPU，各自独立运行。用这种思想构建的程序框架，最大的好处是任务很容易裁剪，系统能够做得很复杂。在充分考虑单片机中断特性（在哪里中断就返回到哪里）后，实际可行的任务切换如图 4B 所示，定时中断可能发生在任务调度，随机任务执行的任何时候，图中最大的框框所示，不管中断在何时发生，它都会正常返回，定时中断所产生的影响只在任务调度模块起作用，即依次让不同的任务按不同的节拍就绪。任务调度会按一定的优先级执行就绪任务。

总结:不同的任务需要 CPU 关照的频度，选择最快的那个频度来设定定时器中断的节拍，一般选择 200Hz，或者 100Hz 都可以。

另外再给每个任务设定一个节拍控制计数器 C，也就是定时器每中断多少次后执行任务一次。例如取定时中断节拍为 200Hz，给任务设定的 C=10，则任务执行频度为 200/10=20Hz，如果是数码管扫描，按 40Hz 不闪烁规律，则任务节拍控制 计数器 C=5 即可。在程序设计中，C 代表着任务运行的节拍控制参数，我们习惯用 delay 来 描述，不同的任务用 task0，task1……来描述。

下面我们来用代码实现以上多任务程序设计思想。
首先是任务切换

```
while(1)
{
 if(task_delay[0]==0)   task0();  //task0 就绪，
 if(task_delay[1]==0)   task1();  //task1 就绪，
 ……
}
```

很显然，执行任务的条件是任务延时量 task_delay=0，那么任务延时量谁来控制呢？定时器啊！定时器中断对任务延时量减一直到归零，标志任务就绪。当没有任务就绪时，任务切换本身就是一个 Idle 任务。

```
void timer0(void) interrupt 1
{
 if(task_delay[0]) task_delay[0]--;
 if(task_delay[1]) task_delay[1]--;
 ……
}
```

例如 timer0 的中断节拍为 200Hz，task0_delay 初值为 10，则 task0()执行频度为 200/10=20Hz。
有了以上基础，我们来设计一个简单多任务程序，进一步深入理解这种程序设计思想。
任务要求：用单片机不同 IO 脚输出 1Hz，5Hz，10Hz，20Hz 方波信号，这个程序很短，将直接给出。

```
#include "reg51.h"
#define TIME_PER_SEC 200   //定义任务时钟频率，200Hz
#define CLOCK 22118400   //定义时钟晶振，单位 Hz
#define MAX_TASK 4     //定义任务数量

extern void task0(void);  //任务声明
extern void task1(void);
extern void task2(void);
extern void task3(void);

sbit f1Hz  = P1^0;  //端口定义
sbit f5Hz  = P1^1;
sbit f10Hz = P1^2;
sbit f20Hz = P1^3;

unsigned char task_delay[4];  //任务延时变量定义
```

```
//定时器 0 初始化
void timer0_init(void)
{
   unsigned char i;
  for(i=0;i<MAX_TASK;i++) task_delay[i]=0;  //任务延时量清零
   TMOD = (TMOD & 0XF0) | 0X01;   //定时器 0 工作在模式 1， 16Bit 定时器模式
   TH0 = 255-CLOCK/TIME_PER_SEC/12/256;
   TL0 = 255-CLOCK/TIME_PER_SEC/12%256;
    TR0 =1;
    ET0 =1;       //开启定时器和中断
}

// 系统 OS 定时中断服务
void timer0(void) interrupt 1
{
   unsigned char i;
   TH0 = 255-CLOCK/TIME_PER_SEC/12/256;
   TL0 = 255-CLOCK/TIME_PER_SEC/12%256;
   for(i=0;i<MAX_TASK;i++) if(task_delay[i]) task_delay[i]--;
//每节拍对任务延时变量减 1，减至 0  后，任务就绪。
}

/*main 主函数*/
void main(void)
{
   timer0_init();
   EA=1;//开总中断
  while(1)
  {
    if(task_delay[0]==0) {task0(); task_delay[0] = TIME_PER_SEC/ 2;}
 //要产生 1hz 信号，翻转周期就是 2Hz，以下同
 if(task_delay[1]==0) {task1(); task_delay[1] = TIME_PER_SEC/10;}
 //要产生 5hz 信号，翻转周期就是 10Hz，以下同
 if(task_delay[2]==0) {task2(); task_delay[2] = TIME_PER_SEC/20;}
 if(task_delay[3]==0) {task3(); task_delay[3] = TIME_PER_SEC/40;}
  }
}

void task0(void)
{
   f1Hz = !f1Hz;
}
```

```
void task1(void)
{
   f5Hz = !f5Hz;
}

void task2(void)
{
   f10Hz = !f10Hz;
}

void task3(void)
{
   f20Hz = !f20Hz;
}
```

仿真效果如图 5 所示。

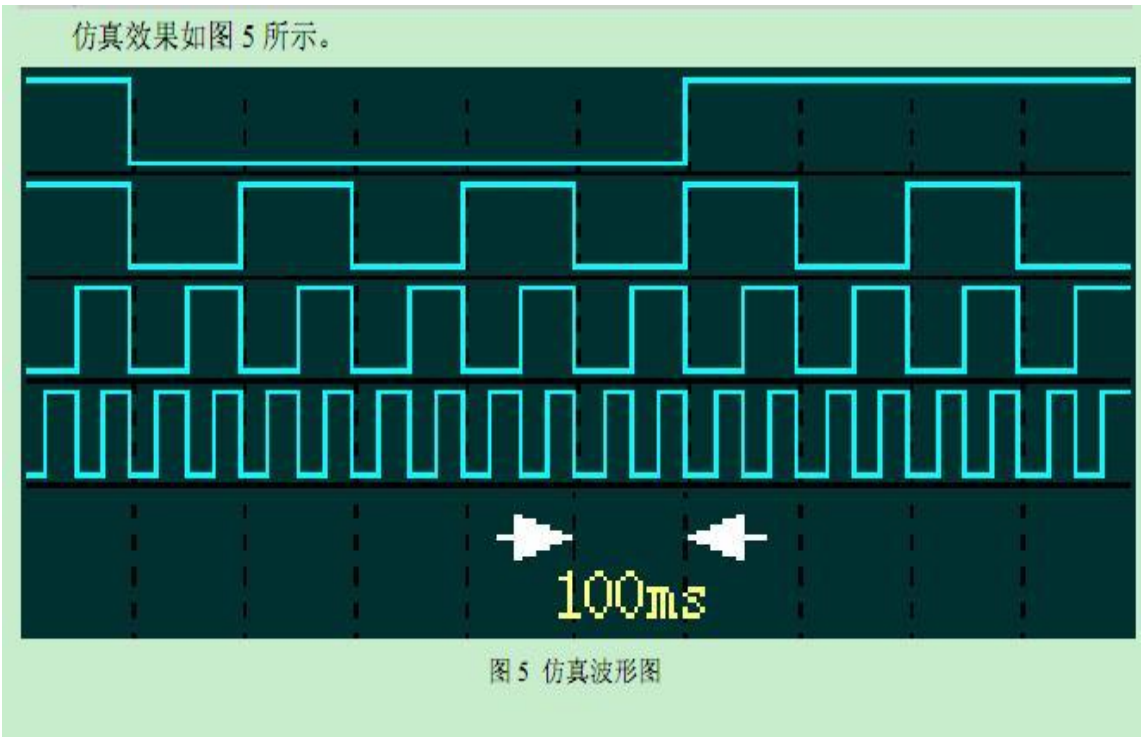


图 5 仿真波形图

同样的程序，同学们可以考虑用图 2 所示的思想设计，看看容易不容易，如果你的程序实现了相同的功能，如果我改变要求，改变信号的频率，你的程序容易修改吗？

要进一步完善这种程序设计思想，有几个问题还需要考虑：

对任务本身有什么要求？

不同任务之间有没有优先级？（不同的事情总有个轻重缓急吧！）

任务间如何延时？

……
为了回答这些问题，下面我们来分析 CPU 的运行情况。

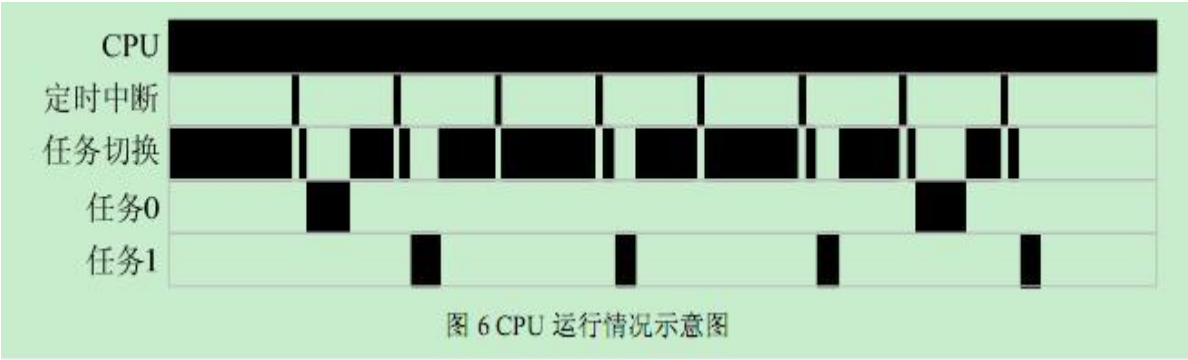


图 6 CPU 运行情况示意图

图 6 CPU 运行情况示意图

CPU 运行情况如图 6 所示，黑色区域表示 CPU 进程，系统启动后， CPU 将无休止的运行， CPU 资源将如何分配呢？程序首先进入“任务切换”进程，如果当前没有任务就绪，就在任务切换进程循环（也可以理解为空闲进程），定时中断将 CPU 当前进程打断，在定时中断进程可能让某些任务就绪，中断返回任务切换进程，很快会进入就绪任务 0，CPU“关照”完任务 0，再次回到任务切换进程，如果还有其它任务就绪，还会再次进入其它任务，没有任务就循环等待，定时中断会不断让新的任务就绪，CPU 也会不断进入任务“关照” 。这样不同的任务就会获得不同的 CPU 资源，每一个任务都像是拥有一个独立的 CPU 为之服务。从这种进程切换我们可以看出，在定时中断和任务切换过程中，额外的占用了一些 CPU 资源， 这就是定时中断频度不宜太快， 否则将大大降低 CPU 的有效资源率， 当然太慢也不行。另外就是 CPU 每次关照任务的时间不能太长，如果超过一个中断周期，就会影响到其它任务的实时性。所谓的实时性就是按定时中断设定的节拍，准时得到 CPU 关照。这样，每一个子任务就必须简单，每次“关照”时间最好不要超过定时中断节拍周期（5ms 或 10ms，初学者要对 ms 有一个概念，机器周期为 us 级的单片机，1ms 可以执行上千条指令，对于像数码管扫描，键盘扫描，LCD 显示等常规任务都是绰绰有余的，只是遇到大型计算，数据排序就显得短了）关于任务优先级的问题：一个复杂系统，多个任务之间总有“轻重缓急”之区别，那些需要严格实时的任务通常用中断实现，中断能够保证第一时间相应，我们这里讨论的不是那种实时概念，是指在最大允许时差内能够得到 CPU“关照” ，例如键盘扫描，为了保证较好的操作效果，快的/慢的/长的/短的（不同人按键不一样）都能够正确识别，这就要保证足够的扫描速度，这种扫描速度对不同的按键最好均等，如果我们按 50Hz 来设计，那么就要保证键盘扫描速度在任何情况下都能够做到 50Hz 扫描频度，不会因为某个新任务的开启而被 破坏，如果确实有新的任务有可能破坏这个 50Hz 扫描频度，我们就应该在优先级安排上让键盘扫描优先级高于那个可能影响键盘扫描的任务。这里体现的就是当同时多个任务就绪时，最先执行哪个的问题，任务调度时要优先执行级别高的任务。 关于“长”任务的问题：有些任务虽然很独立，但完成一次任务执行需要很长时间，例如 DS18B20，从复位初始化到读回温度值，最长接近 1s，这主要是 DS18B20 温度传感器完成一次温度转换需要 500 到 750ms，这个时间对 CPU 而言，简直是太长了，就像一件事情需要我们人等待 10 年一样，显然这样的任务是其它任务所耽搁不起的。像类似 DS18B20 这样的器件（不少 ADC 也是这样） ，怎么设计任务体解决“长”的问题。进一步研究这些器件发现，真正需要 CPU“关照”它们的时间并不长，关键是等待结果要很长时间。解决的办法就是把类似的器件驱动分成多个段：初始化段、启动段、读结果段，而在需要花长时间等待时间段，不要 CPU 关照，允许 CPU 去关照其它任务。

将一个任务分成若干段，确保每段需要 CPU 关照时长小于定时器中断节拍长，这样 CPU 在处理这些长任务时，就不会影响到其它任务的执行。

Easy51RTOS 正是基于以上程序设计思想，总结完善后提出一种耗费资源特别少并且不使用堆栈的多线程操作系统，这个操作系统以纯 C 语言实现，无硬件依赖性，需要单片机的资源极少。起名为 Easy51RTOS，特别适合初学者学习使用。有任务优先级，通过技巧可以任务间延时，缺点是高优先级任务不具有抢占功能，一个具有抢占功能的操作系统，一定要涉及到现场保护与恢复，需要更多的 RAM 资源，涉及到堆栈知识，文件系统将很复杂，初学者学习难度大。

为了便于初学者学习，将代码文件压缩至 4 个文件。

Easy51RTOS.Uv2 Keil 工程文件，KEIL 用户很熟悉的

main.c　main 函数和用户任务 task 函数文件

os_c.c　Easy51RTOS 相关函数文件

os_cfg.h　Easy51RTOS 相关配置参数头文件

文件解读如下：

os_cfg.h

```
#include "reg51.h"
#define TIME_PER_SEC 200   //定义任务时钟频率，200Hz
#define CLOCK 22118400   //定义时钟晶振，单位 Hz
#define MAX_TASK 4     //定义任务数量
//函数变量声明，在需要用以下函数或变量的文件中包含此头文件即可
extern void task0(void);
extern void task1(void);
extern void task2(void);
extern void task3(void);
extern unsigned char task_delay[MAX_TASK];
extern void run(void (*ptask)());
extern void os_timer0_init(void);
```

os_c.c

```
#include "os_cfg.h"
unsigned char task_delay[MAX_TASK];  //定义任务延时量变量
//定时器 0 初始化
void os_timer0_init(void)
{
   unsigned char i;
  for(i=0;i<MAX_TASK;i++) task_delay[i]=0;
   TMOD = (TMOD & 0XF0) | 0X01;   //定时器 0 工作在模式 1，16Bit 定时器模式
   TH0 = 255-CLOCK/TIME_PER_SEC/12/256;
//CRY_OSC,TIME_PER_SEC 在 os_cfg.h 中定义
   TL0 = 255-CLOCK/TIME_PER_SEC/12%256;
    TR0 =1;
    ET0 =1;       //开启定时器和中断
}
```

```
// 系统 OS 定时中断服务
void os_timer0(void) interrupt 1
{
   unsigned char i;
   TH0 = 255-CLOCK/TIME_PER_SEC/12/256;
   TL0 = 255-CLOCK/TIME_PER_SEC/12%256;
   for(i=0;i<MAX_TASK;i++) if(task_delay[i]) task_delay[i]--;
//每节拍对任务延时变量减 1，减至 0 后，任务就绪。
}

//指向函数的指针函数
void run(void (*ptask)())
{
  (*ptask)();
}
```

main.c

```
#include "os_cfg.h"
#define TASK_DELAY0 TIME_PER_SEC/1     //任务执行频度为 1Hz
#define TASK_DELAY1 TIME_PER_SEC/2     //任务执行频度为 2Hz
#define TASK_DELAY2 TIME_PER_SEC/10     //任务执行频度为 10Hz
#define TASK_DELAY3 TIME_PER_SEC/20     //任务执行频度为 20Hz
void (* code task[])() = {task0,task1,task2,task3};  //获得任务 PC 指针

sbit LED0 = P1^0;  //演示用 LED 接口定义
sbit LED1 = P1^1;
sbit LED2 = P1^2;
sbit LED3 = P1^3;

/*main 主函数*/
void main(void)
{
   unsigned char i;
   os_timer0_init();  //节拍发生器定时器初始化
   EA = 1;      //开总中断

  while(1)
  {
    for(i=0;i<MAX_TASK;i++)
    if (task_delay[i]==0) {run(task[i]); break;}  //就绪任务调度
  }  //上一行 break 有特殊作用，详细解释见后文
}
```

```
void task0(void)  //任务 0
{
   LED0 = !LED0;
 task_delay[0] = TASK_DELAY0;
}

void task1(void)  //任务 1
{
   LED1 = !LED1;
 task_delay[1] = TASK_DELAY1;
}

void task2(void)  //任务 2
{
   LED2 = !LED2;
 task_delay[2] = TASK_DELAY2;
}

void task3(void)  //任务内分段设计
{
     static unsigned char state=0;  //定义静态局部变量
 switch (state)
 {
  case 0:
           LED3 = !LED3;
           state = 1;
           task_delay[3] = TASK_DELAY3;
           break;
   case 1:
           LED3 = !LED3;
           state = 2;
           task_delay[3] = TASK_DELAY3*2;
           break;
   case 2:
           LED3 = !LED3;
           state = 0;
           task_delay[3] = TASK_DELAY3*4;
           break;
   default:
            state = 0;
            task_delay[3] = TASK_DELAY3;
            break;
 }
}
```

仿真图如图 8 所示

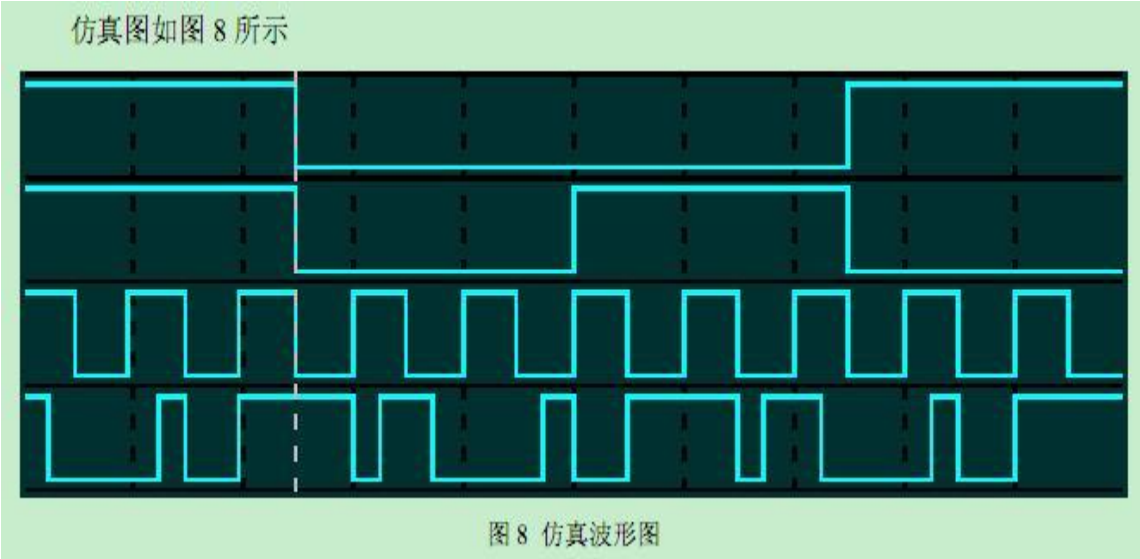


图 8 仿真波形图

图 8 仿真波形图

主程序巧妙实现优先级设定：

```
for(i=0;i<MAX_TASK;i++)
 if (task_delay[i]==0) {run(task[i]); break;} //就绪任务调度
```

这里的 break 将跳出 for 循环，使得每次重新任务调度总是从 task0 开始，就意味着优先级高的任务就绪会先执行。这样 task0 具有最高优先级，task1、task2、task3 优先级依次降低。

特别是 void task3(void)用 switch(state)状态机实现了任务分段，这也是任务内系统延时的一种方法。

九、 综合应用之二——DS1320/DS18B20 应用

对于市面上的大多数 51 单片机开发板来说。ds1302 和 ds18b20 应该是比较常见的两种外围芯片。ds1302 是具有 SPI 总线接口的时钟芯片。ds18b20 则是具有单总线接口的数字温度传感器。下面让我们分别来认识并学会应用这两种芯片。
首先依旧是看 DS1302 的 datasheet 中的相关介绍。

DS1302 是 DALLAS 公司推出的涓流充电时钟芯片，内含有一个实时时钟/日历和 31 字节静态 RAM，通过简单的串行接口与单片机进行通信。实时时钟/日历电路提供秒、分、时、日、日期、月、年的信息，每月的天数和闰年的天数可自动调整，时钟操作可通过 AM/PM 指示决定采用 24 或 12 小时格式。DS1302 与单片机之间能简单地采用同步串行的方式进行通信，仅需用到三个口线：(1)RES(复位)，(2)I/O(数据线)，(3)SCLK(串行时钟)。时钟/RAM 的读/写数据以一个字节或多达 31 个字节的字符组方式通信。DS1302 工作时功耗很低，保持数据和时钟信息时功率小于 1mW。

DS1302 是由 DS1202 改进而来，增加了以下的特性：双电源管脚用于主电源和备份电源供应，Vcc1 为可编程涓流充电电源，附加七个字节存储器。它广泛应用于电话、传真、便携式仪器以及电池供电的仪器仪表等产品领域。下面将主要的性能指标作一综合：

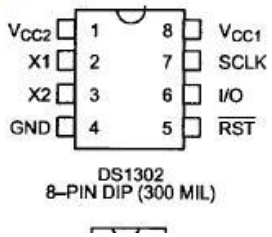
- 实时时钟具有能计算 2100 年之前的秒、分、时、日、日期、星期、月、年的能力，还有闰年调整的能力
- 31×8 位暂存数据存储 RAM
- 串行 I/O 口方式使得管脚数量最少
- 宽范围工作电压：2.0～5.5V
- 工作电流：2.0V 时,小于 300nA
- 读/写时钟或 RAM 数据时，有两种传送方式：单字节传送和多字节传送（字符组方式）
- 8 脚 DIP 封装或可选的 8 脚 SOIC 封装（根据表面装配）
- 简单 3 线接口
- 与 TTL 兼容（Vcc=5V）
- 可选工业级温度范围：-40℃～+85℃

上面是它的一些基本的应用介绍。
下面是它的引脚的描述。

管脚描述

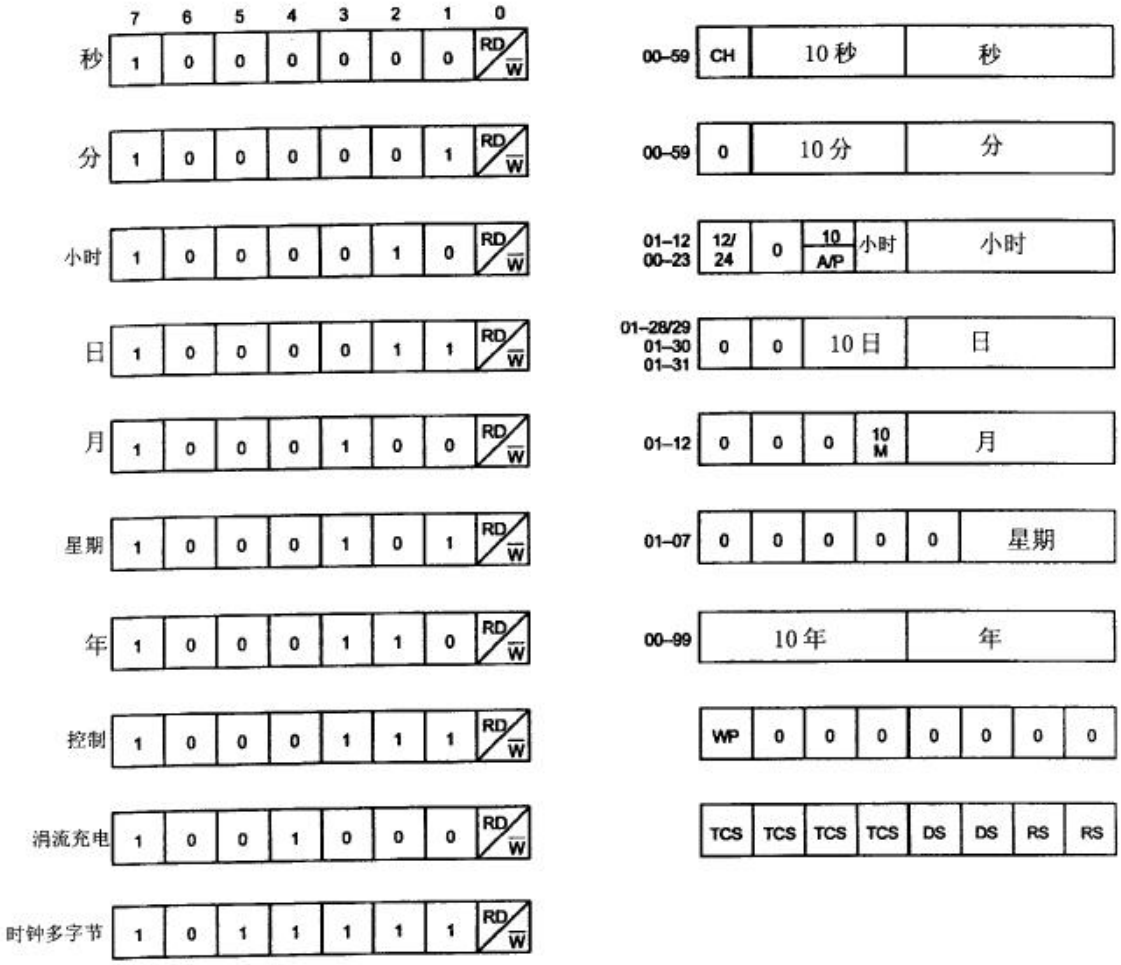
X1，X2	——32.768KHz 晶振管脚
GND	——地
$\overline{RST}$	——复位脚
I/O	——数据输入/输出引脚
SCLK	——串行时钟
Vcc1,Vcc2	——电源供电管脚

管脚配置

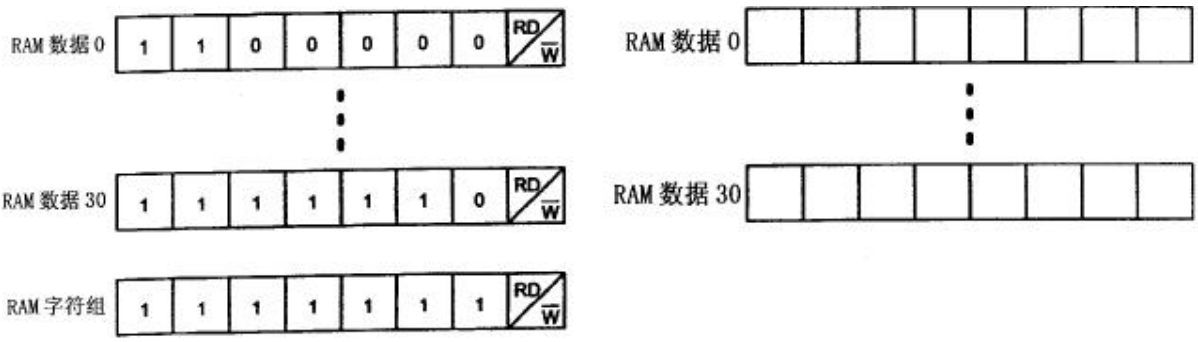


DS1302
8–PIN DIP (300 MIL)

下面是 DS1302 的时钟寄存器。我们要读取的时间数据就是从下面这些数据寄存器中读取出来的。当我们要想调整时间时，可以把时间数据写入到相应的寄存器中就可以了。



这是 DS1302 内部的 31 个 RAM 寄存器。在某些应用场合我们可以应用到。如我们想要做一个带定时功能的闹钟。则可以把闹钟的时间写入到 31 个 RAM 寄存器中的任意几个。当单片机掉电时，只要我们的 DS1302 的备用电池还能工作，那么保存在其中的闹钟数据就不会丢失~~



由于对于这些器件的操作基本上按照数据手册上面提供的时序图和相关命令字来进行操作就可以了。因此在我们应用这些器件的时候一定要对照着手册上面的要求来进行操作。如果觉得还不够放心的话。可以到网上下载一些参考程序。对着手册看别人的程序，看别人的思路是怎么样的。

DS1302 和单片机的连接很简单。只需一根复位线，一根时钟线，一根数据线即可。同时

它本身还需要接一个 32.768KHz 的晶振来提供时钟源。对于晶振的两端可以分别接一个 6PF 左右的电容以提高晶振的精确度。同时可以在第 8 脚接上一个 3.6V 的可充电的电池。当系统正常工作时可以对电池进行涓流充电。当系统掉电时，DS1302 由这个电池提供的能量继续工作。

下面让我们来驱动它。

```
sbit io_DS1302_RST  = P2^0 ;
sbit io_DS1302_IO   = P2^1 ;
sbit io_DS1302_SCLK = P2^2 ;

//-----------------------------------常数宏---------------------------------//
#define DS1302_SECOND_WRITE      0x80       //写时钟芯片的寄存器位置
#define DS1302_MINUTE_WRITE      0x82
#define DS1302_HOUR_WRITE        0x84
#define DS1302_WEEK_WRITE        0x8A
#define DS1302_DAY_WRITE         0x86
#define DS1302_MONTH_WRITE       0x88
#define DS1302_YEAR_WRITE        0x8C

#define DS1302_SECOND_READ       0x81       //读时钟芯片的寄存器位置
#define DS1302_MINUTE_READ       0x83
#define DS1302_HOUR_READ         0x85
#define DS1302_WEEK_READ         0x8B
#define DS1302_DAY_READ          0x87
#define DS1302_MONTH_READ        0x89
#define DS1302_YEAR_READ         0x8D
//-----------------------------------操作宏---------------------------------//
#define DS1302_SCLK_HIGH         io_DS1302_SCLK = 1 ;
#define    DS1302_SCLK_LOW           io_DS1302_SCLK = 0 ;

#define DS1302_IO_HIGH           io_DS1302_IO  = 1 ;
#define DS1302_IO_LOW            io_DS1302_IO  = 0 ;
#define DS1302_IO_READ           io_DS1302_IO

#define DS1302_RST_HIGH          io_DS1302_RST  = 1 ;
#define DS1302_RST_LOW           io_DS1302_RST  = 0 ;
```

```
/*****************************************************
* 保存时间数据的结构体       *
*****************************************************/
struct
{
   uint8 Second ;
   uint8 Minute ;
   uint8 Hour ;
   uint8 Day ;
   uint8 Week ;
   uint8 Month ;
   uint8 Year ;
}CurrentTime ;



/*****************************************************************************
* Function:    static void v_DS1302Write_f( uint8 Content )       *
* Description:向 DS1302 写一个字节的内容           *
* Parameter:uint8 Content :   要写的字节           *
*                    *
*****************************************************************************/

static void v_DS1302Write_f( uint8 Content )
{
   uint8 i ;
   for( i = 8 ; i > 0 ; i-- )
   {
      if( Content & 0x01 )
      {
         DS1302_IO_HIGH
      }
      else
      {
         DS1302_IO_LOW
      }
      Content >>= 1 ;

      DS1302_SCLK_HIGH
      DS1302_SCLK_LOW
   }
}
```

```
/*******************************************************************************
* Function:    static uint8 v_DS1302Read_f( void )         *
* Description: 从 DS1302 当前设定的地址读取一个字节的内容    *
* Parameter:                  *
* Return:    返回读出来的值(uint8)              *
*******************************************************************************/
static uint8 v_DS1302Read_f( void )
{
   uint8 i, ReadValue ;
   DS1302_IO_HIGH
   for( i = 8 ; i > 0 ; i-- )
   {
      ReadValue >>= 1 ;
       if( DS1302_IO_READ )
      {
         ReadValue |= 0x80 ;
      }
      else
      {
         ReadValue &= 0x7f ;
      }

      DS1302_SCLK_HIGH
      DS1302_SCLK_LOW

   }
   return ReadValue ;
}

/*******************************************************************************
* Function:    void v_DS1302WriteByte_f( uint8 Address,  uint8 Content )    *
* Description:  从 DS1302 指定的地址写入一个字节的内容            *
* Parameter:  Address:    要写入数据的地址          *
*    Content:    写入数据的具体值          *
* Return:                                                                      *
*******************************************************************************/
void v_DS1302WriteByte_f( uint8 Address,  uint8 Content )
{
   DS1302_RST_LOW
   DS1302_SCLK_LOW
   DS1302_RST_HIGH

   v_DS1302Write_f( Address ) ;
   v_DS1302Write_f( Content ) ;
```

```
    DS1302_RST_LOW
    DS1302_SCLK_HIGH
}

/*****************************************************************************
* Function:    uint8 v_DS1302ReadByte_f( uint8 Address )                     *
* Description:从 DS1302 指定的地址读出一个字节的内容       *
* Parameter:Address:    要读出数据的地址          *
*                    *
* Return:       指定地址读出的值(uint8)       *
*****************************************************************************/
uint8 v_DS1302ReadByte_f( uint8 Address )
{
    uint8 ReadValue ;
    DS1302_RST_LOW
    DS1302_SCLK_LOW
    DS1302_RST_HIGH

    v_DS1302Write_f( Address ) ;
    ReadValue = v_DS1302Read_f() ;

    DS1302_RST_LOW
    DS1302_SCLK_HIGH
    return ReadValue ;
}
/*****************************************************************************
* Function:    void v_ClockInit_f( void )          *
* Description:初始化写入 DS1302 时钟寄存器的值(主程序中只需调用一次即可) *
* Parameter:
*
*                                    *
* Return:                        *
*****************************************************************************/
void v_ClockInit_f( void )
{
    if( v_DS1302ReadByte_f( 0xc1) != 0xf0 )
    {
        v_DS1302WriteByte_f( 0x8e, 0x00 ) ;    //允许写操作
        v_DS1302WriteByte_f( DS1302_YEAR_WRITE, 0x08 ) ;    //年
        v_DS1302WriteByte_f( DS1302_WEEK_WRITE, 0x04 ) ;    //星期
        v_DS1302WriteByte_f( DS1302_MONTH_WRITE, 0x12 ) ;    //月
        v_DS1302WriteByte_f( DS1302_DAY_WRITE, 0x11 ) ;        //日
        v_DS1302WriteByte_f( DS1302_HOUR_WRITE, 0x13 ) ;    //小时
```

```
        v_DS1302WriteByte_f( DS1302_MINUTE_WRITE, 0x06 ) ;   //分钟
        v_DS1302WriteByte_f( DS1302_SECOND_WRITE, 0x40 ) ;   //秒
        v_DS1302WriteByte_f( 0x90, 0xa5 ) ;                  //充电
        v_DS1302WriteByte_f( 0xc0, 0xf0 ) ;          //判断是否初始化一次标识写入
        v_DS1302WriteByte_f( 0x8e, 0x80 ) ;                  //禁止写操作
    }
}
/*******************************************************************************
* Function:    void v_ClockUpdata_f( void )
* Description:读取时间数据,并保存在结构体 CurrentTime 中
* Parameter:
*
*
* Return:
*******************************************************************************/
void v_ClockUpdata_f( void )
{
    CurrentTime.Second = v_DS1302ReadByte_f( DS1302_SECOND_READ ) ;
    CurrentTime.Minute = v_DS1302ReadByte_f( DS1302_MINUTE_READ ) ;
    CurrentTime.Hour  = v_DS1302ReadByte_f( DS1302_HOUR_READ ) ;
    CurrentTime.Day    = v_DS1302ReadByte_f( DS1302_DAY_READ ) ;
    CurrentTime.Month  = v_DS1302ReadByte_f( DS1302_MONTH_READ ) ;
    CurrentTime.Week  = v_DS1302ReadByte_f( DS1302_WEEK_READ ) ;
    CurrentTime.Year  = v_DS1302ReadByte_f( DS1302_YEAR_READ ) ;
}
```

有了上面的这些函数我们就可以对 DS1302 进行操作了。当我们想要获取当前时间时，只需要调用 v_ClockUpdata_f(void)这个函数即可。读取到的时间数据保存在 CurrentTime 这个结构体中。至于如何把时间数据在数码管或者是液晶屏上显示出来我相信大家应该都会了吧^_^.

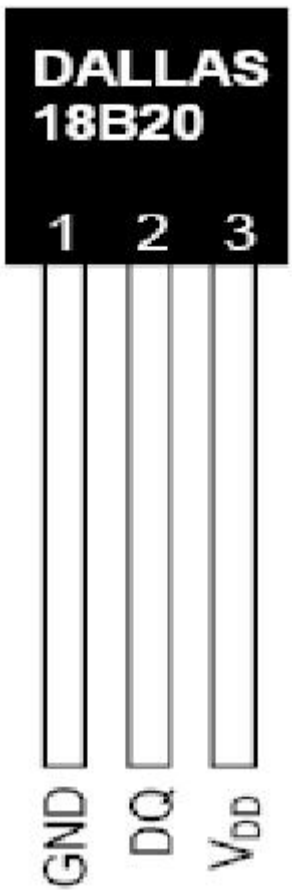
看看显示效果如何~~

下面再让我们看看 DS18B20 吧。

DS18B20 是单总线的数字温度传感器。其与单片机的接口只需要一根数据线即可。当然连线简单意味着软件处理上可能要麻烦一点。下面来看看它的优点：

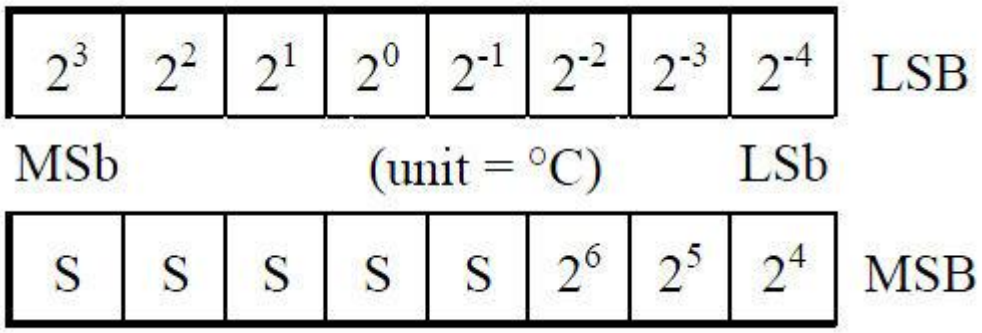
全数字温度转换及输出。
先进的单总线数据通信。
最高 12 位分辨率，精度可达土 0.5 摄氏度。
12 位分辨率时的最大工作周期为 750 毫秒。
可选择寄生工作方式。
检测温度范围为 -55° C ~+125° C (-67° F ~+257° F)
内置 EEPROM，限温报警功能。
64 位光刻 ROM，内置产品序列号，方便多机挂接。
多样封装形式，适应不同硬件系统。

看看它的靓照。外形和我们常用的三极管没有什么两样哦。



DS18B20 的内部存储器分为以下几部分

ROM:存放该器件的编码。前 8 位为单线系列的编码(DS18B20 的编码是 19H)后面 48 位为芯片的唯一序列号。在出场的时候就已经设置好，用户无法更改。最后 8 位是以上 56 位的 CRC 码。



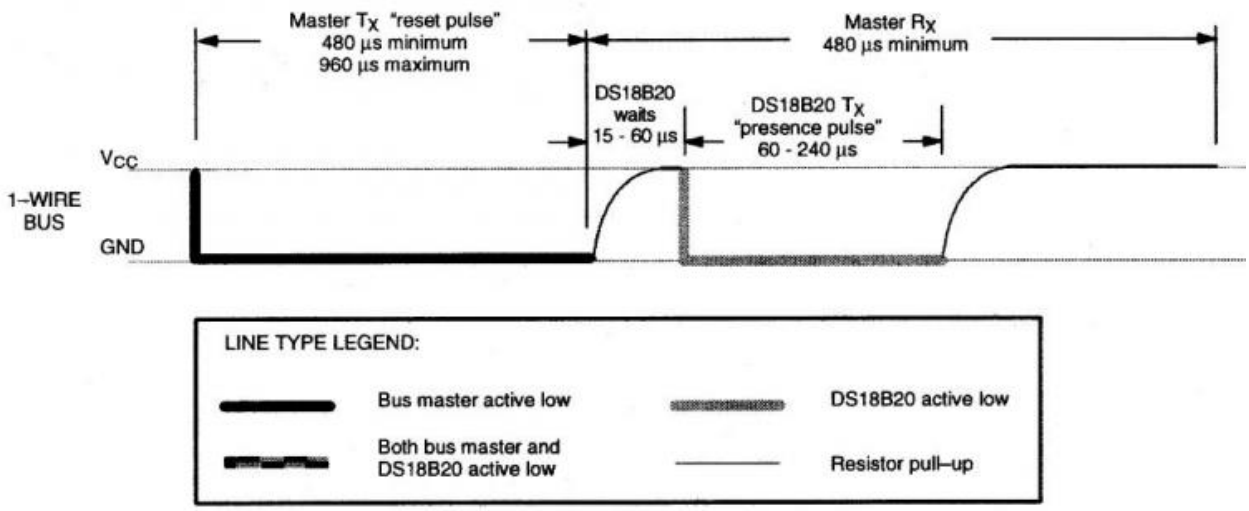
RAM：DS18B20 的内部暂存器共 9 个字节。其中第一个和第二个字节存放转换后的温度值。第二个和第三个字节分别存放高温和低温告警值。(可以用 RAM 指令将其拷贝到 EEPROM 中)第四个字节为配置寄存器。第 5~7 个字节保留。第 9 个字节为前 8 个字节的 CRC 码。

DS18B20 的温度存放如上图所示。其中 S 位符号位。当温度值为负值时，S = 1 ，反之则 S = 0 。我们把得到的温度数据乘上对应的分辨率即可以得到转换后的温度值。

DS18B20 的通讯协议：

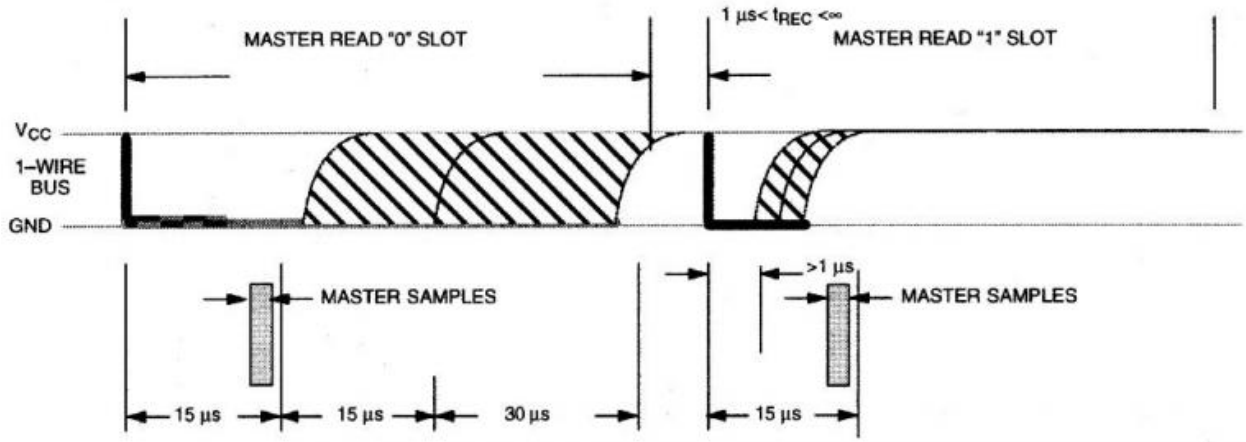
在对 DS18B20 进行读写编程时，必须严格保证读写的时序。否则将无法读取测温结果。根据 DS18B20 的通讯协议，主机控制 DS18B20 完成温度转换必须经过 3 个步骤:每一次读写之前都要对 DS18B20 进行复位，复位成功后发送一条 ROM 指令，最后发送 RAM 指令。这样才能对 DS18B20 进行预定的操作。

复位要求主机将数据线下拉 500us，然后释放，DS18B20 收到信号后等待 16~160us 然后发出 60~240us 的存在低脉冲，主机收到此信号表示复位成功。

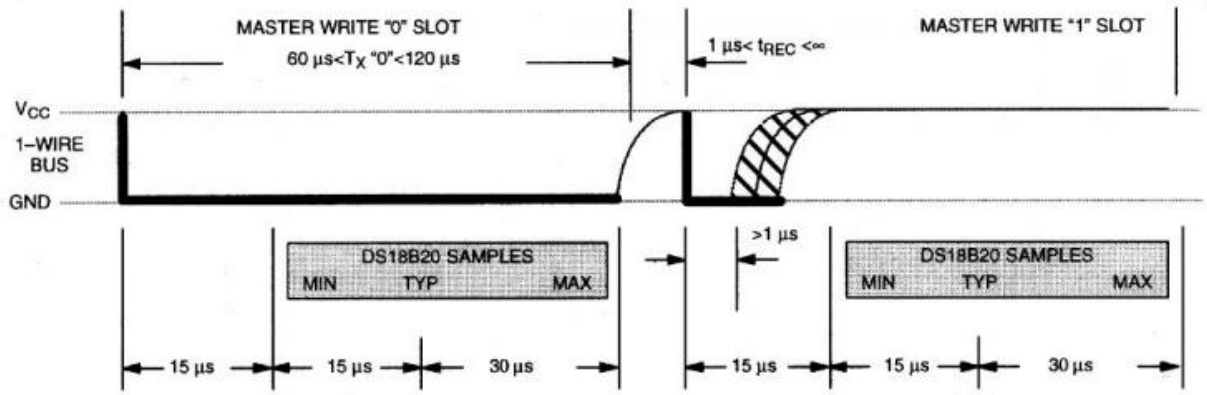


上图即 DS18B20 的复位时序图。

下面是读操作的时序图



这是写操作的时序图



下面让我们来看看它的驱动程序如何写吧。

```
sbit io_DS18B20_DQ  = P2^3 ;
#define DS18B20_DQ_HIGH  io_DS18B20_DQ = 1 ;
#define DS18B20_DQ_LOW  io_DS18B20_DQ = 0 ;
#define DS18B20_DQ_READ  io_DS18B20_DQ
/*********************************************************************
* 保存温度值的数组.依次存放正负标志,温度值十位,个位,和小数位       *
*********************************************************************/
uint8 Temperature[ 4 ] ;
void v_Delay10Us_f( uint16 Count )
{
   while( --Count )
   {
      _nop_();
   }
}
```

```
/*******************************************************************************
* Function:       uint8 v_Ds18b20Init_f( void )                        *
* Description:    初始化 DS18B20                                       *
* Parameter:                                                           *
*                                                                      *
* Return:         返回初始化的结果(0:复位成功  1:复位失败)             *
*******************************************************************************/
uint8 v_Ds18b20Init_f( void )
{
   uint8 Flag ;
   DS18B20_DQ_HIGH         //稍作延时
   v_Delay10Us_f( 3 ) ;
   DS18B20_DQ_LOW          //总线拉低

   v_Delay10Us_f( 80 ) ; //延时大于 480us

   DS18B20_DQ_HIGH     //总线释放

   v_Delay10Us_f( 15 ) ;

   Flag = DS18B20_DQ_READ ;  //如果 Flag 为 0,则复位成功,否则复位失败
   return Flag ;
}
/*******************************************************************************
* Function:       void v_Ds18b20Write_f( uint8 Cmd )                   *
* Description:    向 DS18B20 写命令                                    *
* Parameter:    Cmd:    所要写的命令                                   *
*                                                                      *
* Return:                                                              *
*******************************************************************************/
void v_Ds18b20Write_f( uint8 Cmd )
{
   uint8 i ;
   for( i = 8 ; i > 0 ; i-- )
   {
      DS18B20_DQ_LOW          //拉低总线,开始写时序
      DS18B20_DQ_READ = Cmd & 0x01 ;   //控制字的最低位先送到总线
      v_Delay10Us_f( 5 ) ;          //稍作延时,让 DS18B20 读取总线上的数据
      DS18B20_DQ_HIGH          //拉高总线,1bit 写周期结束
      Cmd >>= 1 ;
   }
}
```

```
/*******************************************************************************
* Function:       uint8 v_Ds18b20Read_f( void )                               *
* Description:    向 DS18B20 读取一个字节的内容                                *
* Parameter:                                                                  *
*                                                                             *
* Return:         读取到的数据                                                *
*******************************************************************************/
uint8 v_Ds18b20Read_f( void )
{
   uint8 ReadValue, i ;
   for( i = 8 ; i > 0 ; i-- )
   {
      DS18B20_DQ_LOW
      ReadValue >>= 1 ;
      DS18B20_DQ_HIGH
      if( DS18B20_DQ_READ == 1 )
      ReadValue |= 0x80 ;
      v_Delay10Us_f( 3 ) ;
   }
   return ReadValue ;
}

/*******************************************************************************
* Function:       uint16 v_Ds18b20ReadTemp_f( void )                          *
* Description:    读取当前的温度数据(只保留了一位小数)                        *
* Parameter:                                                                  *
*                                                                             *
* Return:         读取到的温度值                                              *
*******************************************************************************/
uint16 v_Ds18b20ReadTemp_f( void )
{
   uint8 TempH, TempL ;
   uint16 ReturnTemp ;

/*    if( v_Ds18b20Init_() ) return ;  //复位失败,在这里添加错误处理的代码     */
   v_Ds18b20Init_f() ;           /复位 DS18B20
   v_Ds18b20Write_f( 0xcc ) ;    //跳过 ROM
   v_Ds18b20Write_f( 0x44 ) ;       //启动温度转换
   v_Ds18b20Init_f() ;
   v_Ds18b20Write_f( 0xcc ) ;    //跳过 ROM
   v_Ds18b20Write_f( 0xbe ) ;    //读取 DS18B20 内部的寄存器内容
   TempL = v_Ds18b20Read_f() ;   //读温度值低位（内部 RAM 的第 0 个字节）
   TempH = v_Ds18b20Read_f() ;   //读温度值高位（内部 RAM 的第 1 个字节）
   ReturnTemp = TempH ;
```

```
   ReturnTemp <<= 8 ;
   ReturnTemp |= TempL ;   //温度值放在变量 ReturnTemp 中
   return ReturnTemp ;
}
/*******************************************************************************
* Function:    void v_TemperatureUpdate_f( void )       *
* Description:读取当前的温度数据并转化存放在数组 Temperature(只保留了一位小数） *
* Parameter:
 *
*                        *
* Return:                                                                      *
*******************************************************************************/
void v_TemperatureUpdate_f( void )
{
   uint8 Tflag = 0 ;
   uint16 TempDat ;
   float Temp ;
   TempDat = v_Ds18b20ReadTemp_f() ;
   if( TempDat & 0xf000 )
   {
      Tflag = 1 ;
      TempDat = ~TempDat + 1 ;
   }
   Temp =    TempDat >> 4; // (TempDat * 0.0625 ) 请大家不要用乘以，不知道为什么可以看我下面的贴子
   TempDat = Temp * 10 ; //小数部用可以用查表法，大家有什么好办法来讨论下，呵呵
   Temperature[ 0 ] = Tflag ;        //温度正负标志
   Temperature[ 1 ] = TempDat / 100  + '0' ;     //温度十位值
   Temperature[ 2 ] = TempDat % 100 / 10 + '0' ; //温度个位值
   Temperature[ 3 ] = TempDat % 10       + '0' ;//温度小数位
}
```

如果想获取当前的温度数据，在主函数中调用 v_TemperatureUpdate_f(void)就可以了。温度数据就保存到 Temperature 中去了。至于如何显示，就不用多说了吧~@_@~

时间和温度一起显示出来看看

OK,至此 ds18b20 和 ds1302 的应用告一段落。如果有不懂的，记得多看 datasheet,多交流。

原文地址:http://www.eehome.cn/read.php?tid=14139

一个有关 0.0625℃的运算想到的问题

碰到一哥们号称挺 NB 的嵌入软件工程师，看了他的代码后就欧拉，事情是在一个只有 4K 代码的单片机接 2 个 DS18B20 测温传感器，都知道 DS18B20 输出数据只要乘以 0.0625 就是测量的温度值，这哥们说程序空间怎么也不够，实际上程序只有简单的采集两个 DS18B20 的数据转换成温度值，之后在 1602 液晶上显示，挺简单个程序，怎么也想不通为什么程序空间不够。只读了一下代码发现程序就没动脑子，真的用浮点库把 DS18B20 数据直接乘以 0.0625 了，那程序不超才怪呢，稍微动动脑子也会知道 0.0625 不就是 1/16 吗，把 DS18B20 的数据直接右移 4 位不就是了（当然要注意符号），这右移程序可十分简单还省空间，问题很好解决，空间自然也就够了。　现在想来嵌入处理器确实是进步了，程序空间是越来越大，数据 RAM 空间也越来越大，导致很多人在写程序的时候真的是什么都不顾，借着 C 语言的灵活性真是纵横驰骋，压根也不讲个程序效率和可靠性。正如前些日子见到一孩子用 ARM cortex-m3 处理器给人接活写个便携表的 1024 点 FFT 算法，本身 12 位的 AD 系统，这小家伙直接到网上下载了浮点的 FFT 算法代码就给人加上了，结果整个程序死慢死慢的，人家用户可不买单啊，这时要动动脑子把数据直接变成乘以某个数变成整数后用定点 FFT 处理，之后再把数据除一下不就行了。速度自然也快了，而且也能省下空间。实际当中我们做嵌入软件很多时候犯懒都忽视程序执行效率问题，是都能实现功能，但有时候就是没法谈性能。我几次碰到这样的工程师，直接把传感器的信号放大后进嵌入处理器的 AD，也不看看 AD 数据是否稳定有效，直接就进行 FFT 运算，那 FFT 结果真是热闹，不难看出混叠很严重，于是又机械地在 FFT 基础上再去衍生算法，系统程序越做越大，速度越做越慢。实际上也很简单的事，在传感器放大信号进 AD 之前来一级抗混叠滤波基本也就解决了，大有所谓嵌入软件高手的概念是程序几乎是万能，实在解决不了就换大程序空间更高速的处理器，整个恶性循环。　经常听说现在流行低碳族，我想出色的嵌入软件工程师最容易成为低碳一族，只要让代码高效那处理器频率自然可以灵活降下来，自然耗电也就少了，二氧化碳排放也就少了。想想目前到处都是嵌入处理器，代码条数看来也别有效果。